```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/ansi.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The <ansi.h> header attempts to decide whether the compiler has enough
 * conformance to Standard C for Minix to take advantage of.  If so, the
 * symbol _ANSI is defined (as 31415).  Otherwise _ANSI is not defined
 * here, but it may be defined by applications that want to bend the rules.
 * The magic number in the definition is to inhibit unnecessary bending
 * of the rules.  (For consistency with the new '#ifdef _ANSI" tests in
 * the headers, _ANSI should really be defined as nothing, but that would
 * break many library routines that use "#if _ANSI".)

 * If _ANSI ends up being defined, a macro
 *
 *      _PROTOTYPE(function, params)
 *
 * is defined.  This macro expands in different ways, generating either
 * ANSI Standard C prototypes or old-style K&R (Kernighan & Ritchie)
 * prototypes, as needed.  Finally, some programs use _CONST, _VOIDSTAR etc
 * in such a way that they are portable over both ANSI and K&R compilers.
 * The appropriate macros are defined here.
 */

#ifndef _ANSI_H
#define _ANSI_H

#if __STDC__ == 1
#define _ANSI           31459   /* compiler claims full ANSI conformance */
#endif

#ifdef __GNUC__
#define _ANSI           31459   /* gcc conforms enough even in non-ANSI mode */
#endif

#ifdef _ANSI

/* Keep everything for ANSI prototypes. */
#define _PROTOTYPE(function, params)    function params
#define _ARGS(params)                   params

#define _VOIDSTAR       void *
#define _VOID           void
#define _CONST          const
#define _VOLATILE       volatile
#define _SIZET          size_t

#else

/* Throw away the parameters for K&R prototypes. */
#define _PROTOTYPE(function, params)    function()
#define _ARGS(params)                   ()

#define _VOIDSTAR       void *
#define _VOID           void
#define _CONST
#define _VOLATILE
#define _SIZET          int

```

```
#endif /* _ANSI */

#endif /* ANSI_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/limits.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <limits.h> header defines some basic sizes, both of the language types
 * (e.g., the number of bits in an integer), and of the operating system (e.g.
 * the number of characters in a file name.
 */

#ifndef _LIMITS_H
#define _LIMITS_H

/* Definitions about chars (8 bits in MINIX, and signed). */
#define CHAR_BIT           8    /* # bits in a char */
#define CHAR_MIN        -128    /* minimum value of a char */
#define CHAR_MAX         127    /* maximum value of a char */
#define SCHAR_MIN       -128    /* minimum value of a signed char */
#define SCHAR_MAX        127    /* maximum value of a signed char */
#define UCHAR_MAX        255    /* maximum value of an unsigned char */
#define MB_LEN_MAX         1    /* maximum length of a multibyte char */

/* Definitions about shorts (16 bits in MINIX). */
#define SHRT_MIN  (-32767-1)    /* minimum value of a short */
#define SHRT_MAX       32767    /* maximum value of a short */
#define USHRT_MAX     0xFFFF    /* maximum value of unsigned short */

/* _EM_WSIZE is a compiler-generated symbol giving the word size in bytes. */
#if _EM_WSIZE == 2
#define INT_MIN   (-32767-1)    /* minimum value of a 16-bit int */
#define INT_MAX        32767    /* maximum value of a 16-bit int */
#define UINT_MAX      0xFFFF    /* maximum value of an unsigned 16-bit int */
#endif

#if _EM_WSIZE == 4
#define INT_MIN (-2147483647-1) /* minimum value of a 32-bit int */
#define INT_MAX   2147483647    /* maximum value of a 32-bit int */
#define UINT_MAX  0xFFFFFFFF    /* maximum value of an unsigned 32-bit int */
#endif

/*Definitions about longs (32 bits in MINIX). */
#define LONG_MIN (-2147483647L-1)/* minimum value of a long */
#define LONG_MAX  2147483647L   /* maximum value of a long */
#define ULONG_MAX 0xFFFFFFFFL   /* maximum value of an unsigned long */

/* Minimum sizes required by the POSIX P1003.1 standard (Table 2-3). */
#ifdef _POSIX_SOURCE            /* these are only visible for POSIX */
#define _POSIX_ARG_MAX    4096  /* exec() may have 4K worth of args */
#define _POSIX_CHILD_MAX     6  /* a process may have 6 children */
#define _POSIX_LINK_MAX      8  /* a file may have 8 links */
#define _POSIX_MAX_CANON   255  /* size of the canonical input queue */
#define _POSIX_MAX_INPUT   255  /* you can type 255 chars ahead */
#define _POSIX_NAME_MAX     14  /* a file name may have 14 chars */
#define _POSIX_NGROUPS_MAX   0  /* supplementary group IDs are optional */
#define _POSIX_OPEN_MAX     16  /* a process may have 16 files open */
```

```
#define _POSIX_PATH_MAX    255  /* a pathname may contain 255 chars */
#define _POSIX_PIPE_BUF    512  /* pipes writes of 512 bytes must be atomic */
#define _POSIX_STREAM_MAX    8  /* at least 8 FILEs can be open at once */
#define _POSIX_TZNAME_MAX    3  /* time zone names can be at least 3 chars */
#define _POSIX_SSIZE_MAX 32767  /* read() must support 32767 byte reads */

/* Values actually implemented by MINIX (Tables 2-4, 2-5, 2-6, and 2-7). */
/* Some of these old names had better be defined when not POSIX. */
#define _NO_LIMIT        100    /* arbitrary number; limit not enforced */

#define NGROUPS_MAX        0    /* supplemental group IDs not available */
#if _EM_WSIZE > 2
#define ARG_MAX        16384    /* # bytes of args + environ for exec() */
#else
#define ARG_MAX         4096    /* args + environ on small machines */
#endif
#define CHILD_MAX  _NO_LIMIT    /* MINIX does not limit children */
#define OPEN_MAX          20    /* # open files a process may have */
#define LINK_MAX         127    /* # links a file may have */
#define MAX_CANON        255    /* size of the canonical input queue */
#define MAX_INPUT        255    /* size of the type-ahead buffer */
#define NAME_MAX          14    /* # chars in a file name */
#define PATH_MAX         255    /* # chars in a path name */
#define PIPE_BUF        7168    /* # bytes in atomic write to a pipe */
#define STREAM_MAX        20    /* must be the same as FOPEN_MAX in stdio.h */
#define TZNAME_MAX         3    /* maximum bytes in a time zone name is 3 */
#define SSIZE_MAX      32767    /* max defined byte count for read() */

#endif /* _POSIX_SOURCE */

#endif /* _LIMITS_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/errno.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The <errno.h> header defines the numbers of the various errors that can
 * occur during program execution.  They are visible to user programs and
 * should be small positive integers.  However, they are also used within
 * MINIX, where they must be negative.  For example, the READ system call is
 * executed internally by calling do_read().  This function returns either a
 * (negative) error number or a (positive) number of bytes actually read.
 *
 * To solve the problem of having the error numbers be negative inside the
 * the system and positive outside, the following mechanism is used.  All the
 * definitions are are the form:
 *
 *      #define EPERM           (_SIGN 1)
 *
 * If the macro _SYSTEM is defined, then  _SIGN is set to "-", otherwise it is
 * set to "".  Thus when compiling the operating system, the  macro _SYSTEM
 * will be defined, setting EPERM to (- 1), whereas when when this
 * file is included in an ordinary user program, EPERM has the value ( 1).
 */

#ifndef _ERRNO_H                /* check if <errno.h> is already included */
```

```
#define _ERRNO_H                /* it is not included; note that fact */

/* Now define _SIGN as "" or "-" depending on _SYSTEM. */
#ifdef _SYSTEM
#   define _SIGN         -
#   define OK            0
#else
#   define _SIGN
#endif

extern int errno;                 /* place where the error numbers go */

/* Here are the numerical values of the error numbers. */
#define _NERROR               70  /* number of errors */

#define EGENERIC      (_SIGN 99)  /* generic error */
#define EPERM         (_SIGN  1)  /* operation not permitted */
#define ENOENT        (_SIGN  2)  /* no such file or directory */
#define ESRCH         (_SIGN  3)  /* no such process */
#define EINTR         (_SIGN  4)  /* interrupted function call */
#define EIO           (_SIGN  5)  /* input/output error */
#define ENXIO         (_SIGN  6)  /* no such device or address */
#define E2BIG         (_SIGN  7)  /* arg list too long */
#define ENOEXEC       (_SIGN  8)  /* exec format error */
#define EBADF         (_SIGN  9)  /* bad file descriptor */
#define ECHILD        (_SIGN 10)  /* no child process */
#define EAGAIN        (_SIGN 11)  /* resource temporarily unavailable */
#define ENOMEM        (_SIGN 12)  /* not enough space */
#define EACCES        (_SIGN 13)  /* permission denied */
#define EFAULT        (_SIGN 14)  /* bad address */
#define ENOTBLK       (_SIGN 15)  /* Extension: not a block special file */
#define EBUSY         (_SIGN 16)  /* resource busy */
#define EEXIST        (_SIGN 17)  /* file exists */
#define EXDEV         (_SIGN 18)  /* improper link */
#define ENODEV        (_SIGN 19)  /* no such device */
#define ENOTDIR       (_SIGN 20)  /* not a directory */
#define EISDIR        (_SIGN 21)  /* is a directory */
#define EINVAL        (_SIGN 22)  /* invalid argument */
#define ENFILE        (_SIGN 23)  /* too many open files in system */
#define EMFILE        (_SIGN 24)  /* too many open files */
#define ENOTTY        (_SIGN 25)  /* inappropriate I/O control operation */
#define ETXTBSY       (_SIGN 26)  /* no longer used */
#define EFBIG         (_SIGN 27)  /* file too large */
#define ENOSPC        (_SIGN 28)  /* no space left on device */
#define ESPIPE        (_SIGN 29)  /* invalid seek */
#define EROFS         (_SIGN 30)  /* read-only file system */
#define EMLINK        (_SIGN 31)  /* too many links */
#define EPIPE         (_SIGN 32)  /* broken pipe */
#define EDOM          (_SIGN 33)  /* domain error       (from ANSI C std) */
#define ERANGE        (_SIGN 34)  /* result too large   (from ANSI C std) */
#define EDEADLK       (_SIGN 35)  /* resource deadlock avoided */
#define ENAMETOOLONG  (_SIGN 36)  /* file name too long */
#define ENOLCK        (_SIGN 37)  /* no locks available */
#define ENOSYS        (_SIGN 38)  /* function not implemented */
#define ENOTEMPTY     (_SIGN 39)  /* directory not empty */

/* The following errors relate to networking. */
#define EPACKSIZE     (_SIGN 50)  /* invalid packet size for some protocol */
#define EOUTOFBUFS    (_SIGN 51)  /* not enough buffers left */
#define EBADIOCTL     (_SIGN 52)  /* illegal ioctl for device */
```

```c
#define EBADMODE      (_SIGN 53)  /* badmode in ioctl */
#define EWOULDBLOCK   (_SIGN 54)
#define EBADDEST      (_SIGN 55)  /* not a valid destination address */
#define EDSTNOTRCH    (_SIGN 56)  /* destination not reachable */
#define EISCONN       (_SIGN 57)  /* all ready connected */
#define EADDRINUSE    (_SIGN 58)  /* address in use */
#define ECONNREFUSED  (_SIGN 59)  /* connection refused */
#define ECONNRESET    (_SIGN 60)  /* connection reset */
#define ETIMEDOUT     (_SIGN 61)  /* connection timed out */
#define EURG          (_SIGN 62)  /* urgent data present */
#define ENOURG        (_SIGN 63)  /* no urgent data present */
#define ENOTCONN      (_SIGN 64)  /* no connection (yet or anymore) */
#define ESHUTDOWN     (_SIGN 65)  /* a write call to a shutdown connection */
#define ENOCONN       (_SIGN 66)  /* no such connection */

/* The following are not POSIX errors, but they can still happen. */
#define ELOCKED      (_SIGN 101)  /* can’t send message */
#define EBADCALL     (_SIGN 102)  /* error on send/receive */

/* The following error codes are generated by the kernel itself. */
#ifdef _SYSTEM
#define E_BAD_DEST     -1001    /* destination address illegal */
#define E_BAD_SRC      -1002    /* source address illegal */
#define E_TRY_AGAIN    -1003    /* can’t send-- tables full */
#define E_OVERRUN      -1004    /* interrupt for task that is not waiting */
#define E_BAD_BUF      -1005    /* message buf outside caller’s addr space */
#define E_TASK         -1006    /* can’t send to task */
#define E_NO_MESSAGE   -1007    /* RECEIVE failed: no message present */
#define E_NO_PERM      -1008    /* ordinary users can’t send to tasks */
#define E_BAD_FCN      -1009    /* only valid fcns are SEND, RECEIVE, BOTH */
#define E_BAD_ADDR     -1010    /* bad address given to utility routine */
#define E_BAD_PROC     -1011    /* bad proc number given to utility */
#endif /* _SYSTEM */

#endif /* _ERRNO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/unistd.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```c
/* The <unistd.h> header contains a few miscellaneous manifest constants. */

#ifndef _UNISTD_H
#define _UNISTD_H

/* POSIX requires size_t and ssize_t in <unistd.h> and elsewhere. */
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

#ifndef _SSIZE_T
#define _SSIZE_T
typedef int ssize_t;
#endif
```

```
/* Values used by access().  POSIX Table 2-8. */
#define F_OK               0    /* test if file exists */
#define X_OK               1    /* test if file is executable */
#define W_OK               2    /* test if file is writable */
#define R_OK               4    /* test if file is readable */

/* Values used for whence in lseek(fd, offset, whence).  POSIX Table 2-9. */
#define SEEK_SET           0    /* offset is absolute  */
#define SEEK_CUR           1    /* offset is relative to current position */
#define SEEK_END           2    /* offset is relative to end of file */

/* This value is required by POSIX Table 2-10. */
#define _POSIX_VERSION 199009L  /* which standard is being conformed to */

/* These three definitions are required by POSIX Sec. 8.2.1.2. */
#define STDIN_FILENO       0    /* file descriptor for stdin */
#define STDOUT_FILENO      1    /* file descriptor for stdout */
#define STDERR_FILENO      2    /* file descriptor for stderr */

#ifdef _MINIX
/* How to exit the system. */
#define RBT_HALT           0
#define RBT_REBOOT         1
#define RBT_PANIC          2    /* for servers */
#define RBT_MONITOR        3    /* let the monitor do this */
#define RBT_RESET          4    /* hard reset the system */
#endif

/* NULL must be defined in <unistd.h> according to POSIX Sec. 2.7.1. */
#define NULL    ((void *)0)

/* The following relate to configurable system variables. POSIX Table 4-2. */
#define _SC_ARG_MAX             1
#define _SC_CHILD_MAX           2
#define _SC_CLOCKS_PER_SEC      3
#define _SC_CLK_TCK             3
#define _SC_NGROUPS_MAX         4
#define _SC_OPEN_MAX            5
#define _SC_JOB_CONTROL         6
#define _SC_SAVED_IDS           7
#define _SC_VERSION             8
#define _SC_STREAM_MAX          9
#define _SC_TZNAME_MAX         10

/* The following relate to configurable pathname variables. POSIX Table 5-2. */
#define _PC_LINK_MAX            1       /* link count */
#define _PC_MAX_CANON           2       /* size of the canonical input queue */
#define _PC_MAX_INPUT           3       /* type-ahead buffer size */
#define _PC_NAME_MAX            4       /* file name size */
#define _PC_PATH_MAX            5       /* pathname size */
#define _PC_PIPE_BUF            6       /* pipe size */
#define _PC_NO_TRUNC            7       /* treatment of long name components */
#define _PC_VDISABLE            8       /* tty disable */
#define _PC_CHOWN_RESTRICTED    9       /* chown restricted or not */

/* POSIX defines several options that may be implemented or not, at the
 * implementer's whim.  This implementer has made the following choices:
 *
 * _POSIX_JOB_CONTROL       not defined:        no job control
```

```
 * _POSIX_SAVED_IDS         not defined:        no saved uid/gid
 * _POSIX_NO_TRUNC          defined as -1:      long path names are truncated
 * _POSIX_CHOWN_RESTRICTED  defined:            you can't give away files
 * _POSIX_VDISABLE          defined:            tty functions can be disabled
 */
#define _POSIX_NO_TRUNC       (-1)
#define _POSIX_CHOWN_RESTRICTED  1

/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( void _exit, (int _status)                                   );
_PROTOTYPE( int access, (const char *_path, int _amode)                 );
_PROTOTYPE( unsigned int alarm, (unsigned int _seconds)                 );
_PROTOTYPE( int chdir, (const char *_path)                              );
_PROTOTYPE( int chown, (const char *_path, Uid_t _owner, Gid_t _group)  );
_PROTOTYPE( int close, (int _fd)                                        );
_PROTOTYPE( char *ctermid, (char *_s)                                   );
_PROTOTYPE( char *cuserid, (char *_s)                                   );
_PROTOTYPE( int dup, (int _fd)                                          );
_PROTOTYPE( int dup2, (int _fd, int _fd2)                               );
_PROTOTYPE( int execl, (const char *_path, const char *_arg, ...)       );
_PROTOTYPE( int execle, (const char *_path, const char *_arg, ...)      );
_PROTOTYPE( int execlp, (const char *_file, const char *arg, ...)       );
_PROTOTYPE( int execv, (const char *_path, char *const _argv[])         );
_PROTOTYPE( int execve, (const char *_path, char *const _argv[],
                                                char *const _envp[])    );
_PROTOTYPE( int execvp, (const char *_file, char *const _argv[])        );
_PROTOTYPE( pid_t fork, (void)                                          );
_PROTOTYPE( long fpathconf, (int _fd, int _name)                        );
_PROTOTYPE( char *getcwd, (char *_buf, size_t _size)                    );
_PROTOTYPE( gid_t getegid, (void)                                       );
_PROTOTYPE( uid_t geteuid, (void)                                       );
_PROTOTYPE( gid_t getgid, (void)                                        );
_PROTOTYPE( int getgroups, (int _gidsetsize, gid_t _grouplist[])        );
_PROTOTYPE( char *getlogin, (void)                                      );
_PROTOTYPE( pid_t getpgrp, (void)                                       );
_PROTOTYPE( pid_t getpid, (void)                                        );
_PROTOTYPE( pid_t getppid, (void)                                       );
_PROTOTYPE( uid_t getuid, (void)                                        );
_PROTOTYPE( int isatty, (int _fd)                                       );
_PROTOTYPE( int link, (const char *_existing, const char *_new)         );
_PROTOTYPE( off_t lseek, (int _fd, off_t _offset, int _whence)          );
_PROTOTYPE( long pathconf, (const char *_path, int _name)               );
_PROTOTYPE( int pause, (void)                                           );
_PROTOTYPE( int pipe, (int _fildes[2])                                  );
_PROTOTYPE( ssize_t read, (int _fd, void *_buf, size_t _n)              );
_PROTOTYPE( int rmdir, (const char *_path)                              );
_PROTOTYPE( int setgid, (Gid_t _gid)                                    );
_PROTOTYPE( int setpgid, (pid_t _pid, pid_t _pgid)                      );
_PROTOTYPE( pid_t setsid, (void)                                        );
_PROTOTYPE( int setuid, (Uid_t _uid)                                    );
_PROTOTYPE( unsigned int sleep, (unsigned int _seconds)                 );
_PROTOTYPE( long sysconf, (int _name)                                   );
_PROTOTYPE( pid_t tcgetpgrp, (int _fd)                                  );
_PROTOTYPE( int tcsetpgrp, (int _fd, pid_t _pgrp_id)                    );
_PROTOTYPE( char *ttyname, (int _fd)                                    );
_PROTOTYPE( int unlink, (const char *_path)                             );
```

```
_PROTOTYPE( ssize_t write, (int _fd, const void *_buf, size_t _n)       );

#ifdef _MINIX
_PROTOTYPE( int brk, (char *_addr)                                        );
_PROTOTYPE( int chroot, (const char *_name)                               );
_PROTOTYPE( int mknod, (const char *_name, Mode_t _mode, Dev_t _addr)   );
_PROTOTYPE( int mknod4, (const char *_name, Mode_t _mode, Dev_t _addr,
            long _size)                                                   );
_PROTOTYPE( char *mktemp, (char *_template)                              );
_PROTOTYPE( int mount, (char *_spec, char *_name, int _flag)            );
_PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );
_PROTOTYPE( char *sbrk, (int _incr)                                      );
_PROTOTYPE( int sync, (void)                                              );
_PROTOTYPE( int umount, (const char *_name)                              );
_PROTOTYPE( int reboot, (int _how, ...)                                  );
_PROTOTYPE( int gethostname, (char *_hostname, size_t _len)             );
_PROTOTYPE( int getdomainname, (char *_domain, size_t _len)             );
_PROTOTYPE( int ttyslot, (void)                                           );
_PROTOTYPE( int fttyslot, (int _fd)                                       );
_PROTOTYPE( char *crypt, (const char *_key, const char *_salt)          );
#endif

#endif /* _UNISTD_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/string.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <string.h> header contains prototypes for the string handling
 * functions.
 */

#ifndef _STRING_H
#define _STRING_H

#define NULL    ((void *)0)

#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;    /* type returned by sizeof */
#endif /*_SIZE_T */

/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( void *memchr, (const void *_s, int _c, size_t _n)          );
_PROTOTYPE( int memcmp, (const void *_s1, const void *_s2, size_t _n)   );
_PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t _n)       );
_PROTOTYPE( void *memmove, (void *_s1, const void *_s2, size_t _n)      );
_PROTOTYPE( void *memset, (void *_s, int _c, size_t _n)                 );
_PROTOTYPE( char *strcat, (char *_s1, const char *_s2)                  );
_PROTOTYPE( char *strchr, (const char *_s, int _c)                      );
_PROTOTYPE( int strncmp, (const char *_s1, const char *_s2, size_t _n)  );
_PROTOTYPE( int strcmp, (const char *_s1, const char *_s2)              );
_PROTOTYPE( int strcoll, (const char *_s1, const char *_s2)             );
_PROTOTYPE( char *strcpy, (char *_s1, const char *_s2)                  );
```

```
_PROTOTYPE( size_t strcspn, (const char *_s1, const char *_s2)         );
_PROTOTYPE( char *strerror, (int _errnum)                              );
_PROTOTYPE( size_t strlen, (const char *_s)                            );
_PROTOTYPE( char *strncat, (char *_s1, const char *_s2, size_t _n)     );
_PROTOTYPE( char *strncpy, (char *_s1, const char *_s2, size_t _n)     );
_PROTOTYPE( char *strpbrk, (const char *_s1, const char *_s2)          );
_PROTOTYPE( char *strrchr, (const char *_s, int _c)                    );
_PROTOTYPE( size_t strspn, (const char *_s1, const char *_s2)          );
_PROTOTYPE( char *strstr, (const char *_s1, const char *_s2)           );
_PROTOTYPE( char *strtok, (char *_s1, const char *_s2)                 );
_PROTOTYPE( size_t strxfrm, (char *_s1, const char *_s2, size_t _n)    );

#ifdef _MINIX
/* For backward compatibility. */
_PROTOTYPE( char *index, (const char *_s, int _charwanted)             );
_PROTOTYPE( char *rindex, (const char *_s, int _charwanted)            );
_PROTOTYPE( void bcopy, (const void *_src, void *_dst, size_t _length) );
_PROTOTYPE( int bcmp, (const void *_s1, const void *_s2, size_t _length));
_PROTOTYPE( void bzero, (void *_dst, size_t _length)                   );
_PROTOTYPE( void *memccpy, (char *_dst, const char *_src, int _ucharstop,
                                                    size_t _size)      );
/* BSD functions */
_PROTOTYPE( int strcasecmp, (const char *_s1, const char *_s2)         );
#endif

#endif /* _STRING_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/signal.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <signal.h> header defines all the ANSI and POSIX signals.
 * MINIX supports all the signals required by POSIX. They are defined below.
 * Some additional signals are also supported.
 */

#ifndef _SIGNAL_H
#define _SIGNAL_H

#ifndef _ANSI_H
#include <ansi.h>
#endif

/* Here are types that are closely associated with signal handling. */
typedef int sig_atomic_t;

#ifdef _POSIX_SOURCE
#ifndef _SIGSET_T
#define _SIGSET_T
typedef unsigned long sigset_t;
#endif
#endif

#define _NSIG             16    /* number of signals used */

#define SIGHUP             1    /* hangup */
```

```
#define SIGINT             2    /* interrupt (DEL) */
#define SIGQUIT            3    /* quit (ASCII FS) */
#define SIGILL             4    /* illegal instruction */
#define SIGTRAP            5    /* trace trap (not reset when caught) */
#define SIGABRT            6    /* IOT instruction */
#define SIGIOT             6    /* SIGABRT for people who speak PDP-11 */
#define SIGUNUSED          7    /* spare code */
#define SIGFPE             8    /* floating point exception */
#define SIGKILL            9    /* kill (cannot be caught or ignored) */
#define SIGUSR1           10    /* user defined signal # 1 */
#define SIGSEGV           11    /* segmentation violation */
#define SIGUSR2           12    /* user defined signal # 2 */
#define SIGPIPE           13    /* write on a pipe with no one to read it */
#define SIGALRM           14    /* alarm clock */
#define SIGTERM           15    /* software termination signal from kill */

#define SIGEMT             7    /* obsolete */
#define SIGBUS            10    /* obsolete */

/* POSIX requires the following signals to be defined, even if they are
 * not supported.  Here are the definitions, but they are not supported.
 */
#define SIGCHLD           17    /* child process terminated or stopped */
#define SIGCONT           18    /* continue if stopped */
#define SIGSTOP           19    /* stop signal */
#define SIGTSTP           20    /* interactive stop signal */
#define SIGTTIN           21    /* background process wants to read */
#define SIGTTOU           22    /* background process wants to write */

/* The sighandler_t type is not allowed unless _POSIX_SOURCE is defined. */
#ifdef _POSIX_SOURCE
#define __sighandler_t sighandler_t
#else
typedef void (*__sighandler_t) (int);
#endif

/* Macros used as function pointers. */
#define SIG_ERR    ((__sighandler_t) -1)        /* error return */
#define SIG_DFL    ((__sighandler_t)  0)        /* default signal handling */
#define SIG_IGN    ((__sighandler_t)  1)        /* ignore signal */
#define SIG_HOLD   ((__sighandler_t)  2)        /* block signal */
#define SIG_CATCH  ((__sighandler_t)  3)        /* catch signal */

#ifdef _POSIX_SOURCE
struct sigaction {
  __sighandler_t sa_handler;    /* SIG_DFL, SIG_IGN, or pointer to function */
  sigset_t sa_mask;             /* signals to be blocked during handler */
  int sa_flags;                 /* special flags */
};

/* Fields for sa_flags. */
#define SA_ONSTACK   0x0001     /* deliver signal on alternate stack */
#define SA_RESETHAND 0x0002     /* reset signal handler when signal caught */
#define SA_NODEFER   0x0004     /* don't block signal while catching it */
#define SA_RESTART   0x0008     /* automatic system call restart */
#define SA_SIGINFO   0x0010     /* extended signal handling */
#define SA_NOCLDWAIT 0x0020     /* don't create zombies */
#define SA_NOCLDSTOP 0x0040     /* don't receive SIGCHLD when child stops */

/* POSIX requires these values for use with sigprocmask(2). */
```

```
#define SIG_BLOCK          0    /* for blocking signals */
#define SIG_UNBLOCK        1    /* for unblocking signals */
#define SIG_SETMASK        2    /* for setting the signal mask */
#define SIG_INQUIRE        4    /* for internal use only */
#endif  /* _POSIX_SOURCE */

/* POSIX and ANSI function prototypes. */
_PROTOTYPE( int raise, (int _sig)                                       );
_PROTOTYPE( __sighandler_t signal, (int _sig, __sighandler_t _func)     );

#ifdef _POSIX_SOURCE
_PROTOTYPE( int kill, (pid_t _pid, int _sig)                            );
_PROTOTYPE( int sigaction,
    (int _sig, const struct sigaction *_act, struct sigaction *_oact)   );
_PROTOTYPE( int sigaddset, (sigset_t *_set, int _sig)                   );
_PROTOTYPE( int sigdelset, (sigset_t *_set, int _sig)                   );
_PROTOTYPE( int sigemptyset, (sigset_t *_set)                           );
_PROTOTYPE( int sigfillset, (sigset_t *_set)                            );
_PROTOTYPE( int sigismember, (sigset_t *_set, int _sig)                 );
_PROTOTYPE( int sigpending, (sigset_t *_set)                            );
_PROTOTYPE( int sigprocmask,
            (int _how, const sigset_t *_set, sigset_t *_oset)           );
_PROTOTYPE( int sigsuspend, (const sigset_t *_sigmask)                  );
#endif

#endif /* _SIGNAL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/fcntl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <fcntl.h> header is needed by the open() and fcntl() system calls,
 * which  have a variety of parameters and flags.  They are described here.
 * The formats of the calls to each of these are:
 *
 *      open(path, oflag [,mode])       open a file
 *      fcntl(fd, cmd [,arg])           get or set file attributes
 *
 */

#ifndef _FCNTL_H
#define _FCNTL_H

/* These values are used for cmd in fcntl().  POSIX Table 6-1.  */
#define F_DUPFD            0    /* duplicate file descriptor */
#define F_GETFD            1    /* get file descriptor flags */
#define F_SETFD            2    /* set file descriptor flags */
#define F_GETFL            3    /* get file status flags */
#define F_SETFL            4    /* set file status flags */
#define F_GETLK            5    /* get record locking information */
#define F_SETLK            6    /* set record locking information */
#define F_SETLKW           7    /* set record locking info; wait if blocked */

/* File descriptor flags used for fcntl().  POSIX Table 6-2. */
#define FD_CLOEXEC         1    /* close on exec flag for third arg of fcntl */
```

```c
/* L_type values for record locking with fcntl().  POSIX Table 6-3. */
#define F_RDLCK            1    /* shared or read lock */
#define F_WRLCK            2    /* exclusive or write lock */
#define F_UNLCK            3    /* unlock */

/* Oflag values for open().  POSIX Table 6-4. */
#define O_CREAT        00100    /* creat file if it doesn't exist */
#define O_EXCL         00200    /* exclusive use flag */
#define O_NOCTTY       00400    /* do not assign a controlling terminal */
#define O_TRUNC        01000    /* truncate flag */

/* File status flags for open() and fcntl().  POSIX Table 6-5. */
#define O_APPEND       02000    /* set append mode */
#define O_NONBLOCK     04000    /* no delay */

/* File access modes for open() and fcntl().  POSIX Table 6-6. */
#define O_RDONLY           0    /* open(name, O_RDONLY) opens read only */
#define O_WRONLY           1    /* open(name, O_WRONLY) opens write only */
#define O_RDWR             2    /* open(name, O_RDWR) opens read/write */

/* Mask for use with file access modes.  POSIX Table 6-7. */
#define O_ACCMODE         03    /* mask for file access modes */

/* Struct used for locking.  POSIX Table 6-8. */
struct flock {
  short l_type;                 /* type: F_RDLCK, F_WRLCK, or F_UNLCK */
  short l_whence;               /* flag for starting offset */
  off_t l_start;                /* relative offset in bytes */
  off_t l_len;                  /* size; if 0, then until EOF */
  pid_t l_pid;                  /* process id of the locks' owner */
};


/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( int creat, (const char *_path, Mode_t _mode)                );
_PROTOTYPE( int fcntl, (int _filedes, int _cmd, ...)                    );
_PROTOTYPE( int open,  (const char *_path, int _oflag, ...)             );

#endif /* _FCNTL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/stdlib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```c
/* The <stdlib.h> header defines certain common macros, types, and functions.*/

#ifndef _STDLIB_H
#define _STDLIB_H

/* The macros are NULL, EXIT_FAILURE, EXIT_SUCCESS, RAND_MAX, and MB_CUR_MAX.*/
#define NULL    ((void *)0)

#define EXIT_FAILURE       1    /* standard error return using exit() */
#define EXIT_SUCCESS       0    /* successful return using exit() */
```

```
#define RAND_MAX       32767    /* largest value generated by rand() */
#define MB_CUR_MAX         1    /* max value of multibyte character in MINIX */

typedef struct { int quot, rem; } div_t;
typedef struct { long quot, rem; } ldiv_t;

/* The types are size_t, wchar_t, div_t, and ldiv_t. */
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;    /* type returned by sizeof */
#endif

#ifndef _WCHAR_T
#define _WCHAR_T
typedef char wchar_t;           /* type expanded character set */
#endif

/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( void abort, (void)                                          );
_PROTOTYPE( int abs, (int _j)                                           );
_PROTOTYPE( int atexit, (void (*_func)(void))                           );
_PROTOTYPE( double atof, (const char *_nptr)                            );
_PROTOTYPE( int atoi, (const char *_nptr)                               );
_PROTOTYPE( long atol, (const char *_nptr)                              );
_PROTOTYPE( void *calloc, (size_t _nmemb, size_t _size)                 );
_PROTOTYPE( div_t div, (int _numer, int _denom)                         );
_PROTOTYPE( void exit, (int _status)                                    );
_PROTOTYPE( void free, (void *_ptr)                                     );
_PROTOTYPE( char *getenv, (const char *_name)                           );
_PROTOTYPE( long labs, (long _j)                                        );
_PROTOTYPE( ldiv_t ldiv, (long _numer, long _denom)                     );
_PROTOTYPE( void *malloc, (size_t _size)                                );
_PROTOTYPE( int mblen, (const char *_s, size_t _n)                      );
_PROTOTYPE( size_t mbstowcs, (wchar_t *_pwcs, const char *_s, size_t _n));
_PROTOTYPE( int mbtowc, (wchar_t *_pwc, const char *_s, size_t _n)      );
_PROTOTYPE( int rand, (void)                                            );
_PROTOTYPE( void *realloc, (void *_ptr, size_t _size)                   );
_PROTOTYPE( void srand, (unsigned int _seed)                            );
_PROTOTYPE( double strtod, (const char *_nptr, char **_endptr)          );
_PROTOTYPE( long strtol, (const char *_nptr, char **_endptr, int _base) );
_PROTOTYPE( int system, (const char *_string)                           );
_PROTOTYPE( size_t wcstombs, (char *_s, const wchar_t *_pwcs, size_t _n));
_PROTOTYPE( int wctomb, (char *_s, wchar_t _wchar)                      );
_PROTOTYPE( void *bsearch, (const void *_key, const void *_base,
        size_t _nmemb, size_t _size,
        int (*compar) (const void *, const void *))                     );
_PROTOTYPE( void qsort, (void *_base, size_t _nmemb, size_t _size,
        int (*compar) (const void *, const void *))                     );
_PROTOTYPE( unsigned long int strtoul,
                        (const char *_nptr, char **_endptr, int _base)  );

#ifdef _MINIX
_PROTOTYPE( int putenv, (const char *_name)                             );
_PROTOTYPE(int getopt, (int _argc, char **_argv, char *_opts));
extern char *optarg;
extern int optind, opterr, optopt;
```

```
#endif /* _MINIX */

#endif /* STDLIB_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/termios.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <termios.h> header is used for controlling tty modes. */

#ifndef _TERMIOS_H
#define _TERMIOS_H

typedef unsigned short tcflag_t;
typedef unsigned char cc_t;
typedef unsigned int speed_t;

#define NCCS            20      /* size of cc_c array, some extra space
                                 * for extensions. */

/* Primary terminal control structure. POSIX Table 7-1. */
struct termios {
  tcflag_t c_iflag;             /* input modes */
  tcflag_t c_oflag;             /* output modes */
  tcflag_t c_cflag;             /* control modes */
  tcflag_t c_lflag;             /* local modes */
  speed_t  c_ispeed;            /* input speed */
  speed_t  c_ospeed;            /* output speed */
  cc_t c_cc[NCCS];              /* control characters */
};

/* Values for termios c_iflag bit map.  POSIX Table 7-2. */
#define BRKINT          0x0001  /* signal interrupt on break */
#define ICRNL           0x0002  /* map CR to NL on input */
#define IGNBRK          0x0004  /* ignore break */
#define IGNCR           0x0008  /* ignore CR */
#define IGNPAR          0x0010  /* ignore characters with parity errors */
#define INLCR           0x0020  /* map NL to CR on input */
#define INPCK           0x0040  /* enable input parity check */
#define ISTRIP          0x0080  /* mask off 8th bit */
#define IXOFF           0x0100  /* enable start/stop input control */
#define IXON            0x0200  /* enable start/stop output control */
#define PARMRK          0x0400  /* mark parity errors in the input queue */

/* Values for termios c_oflag bit map.  POSIX Sec. 7.1.2.3. */
#define OPOST           0x0001  /* perform output processing */

/* Values for termios c_cflag bit map.  POSIX Table 7-3. */
#define CLOCAL          0x0001  /* ignore modem status lines */
#define CREAD           0x0002  /* enable receiver */
#define CSIZE           0x000C  /* number of bits per character */
#define         CS5     0x0000  /* if CSIZE is CS5, characters are 5 bits */
#define         CS6     0x0004  /* if CSIZE is CS6, characters are 6 bits */
#define         CS7     0x0008  /* if CSIZE is CS7, characters are 7 bits */
#define         CS8     0x000C  /* if CSIZE is CS8, characters are 8 bits */
#define CSTOPB          0x0010  /* send 2 stop bits if set, else 1 */
#define HUPCL           0x0020  /* hang up on last close */
#define PARENB          0x0040  /* enable parity on output */
```

```
#define PARODD          0x0080  /* use odd parity if set, else even */

/* Values for termios c_lflag bit map.  POSIX Table 7-4. */
#define ECHO            0x0001  /* enable echoing of input characters */
#define ECHOE           0x0002  /* echo ERASE as backspace */
#define ECHOK           0x0004  /* echo KILL */
#define ECHONL          0x0008  /* echo NL */
#define ICANON          0x0010  /* canonical input (erase and kill enabled) */
#define IEXTEN          0x0020  /* enable extended functions */
#define ISIG            0x0040  /* enable signals */
#define NOFLSH          0x0080  /* disable flush after interrupt or quit */
#define TOSTOP          0x0100  /* send SIGTTOU (job control, not implemented*/

/* Indices into c_cc array.  Default values in parentheses. POSIX Table 7-5. */
#define VEOF               0    /* cc_c[VEOF] = EOF char (^D) */
#define VEOL               1    /* cc_c[VEOL] = EOL char (undef) */
#define VERASE             2    /* cc_c[VERASE] = ERASE char (^H) */
#define VINTR              3    /* cc_c[VINTR] = INTR char (DEL) */
#define VKILL              4    /* cc_c[VKILL] = KILL char (^U) */
#define VMIN               5    /* cc_c[VMIN] = MIN value for timer */
#define VQUIT              6    /* cc_c[VQUIT] = QUIT char (^\) */
#define VTIME              7    /* cc_c[VTIME] = TIME value for timer */
#define VSUSP              8    /* cc_c[VSUSP] = SUSP (^Z, ignored) */
#define VSTART             9    /* cc_c[VSTART] = START char (^S) */
#define VSTOP             10    /* cc_c[VSTOP] = STOP char (^Q) */

#define _POSIX_VDISABLE   (cc_t)0xFF    /* You can't even generate this
                                         * character with 'normal' keyboards.
                                         * But some language specific keyboards
                                         * can generate 0xFF. It seems that all
                                         * 256 are used, so cc_t should be a
                                         * short...
                                         */

/* Values for the baud rate settings.  POSIX Table 7-6. */
#define B0              0x0000  /* hang up the line */
#define B50             0x1000  /* 50 baud */
#define B75             0x2000  /* 75 baud */
#define B110            0x3000  /* 110 baud */
#define B134            0x4000  /* 134.5 baud */
#define B150            0x5000  /* 150 baud */
#define B200            0x6000  /* 200 baud */
#define B300            0x7000  /* 300 baud */
#define B600            0x8000  /* 600 baud */
#define B1200           0x9000  /* 1200 baud */
#define B1800           0xA000  /* 1800 baud */
#define B2400           0xB000  /* 2400 baud */
#define B4800           0xC000  /* 4800 baud */
#define B9600           0xD000  /* 9600 baud */
#define B19200          0xE000  /* 19200 baud */
#define B38400          0xF000  /* 38400 baud */

/* Optional actions for tcsetattr().  POSIX Sec. 7.2.1.2. */
#define TCSANOW            1    /* changes take effect immediately */
#define TCSADRAIN          2    /* changes take effect after output is done */
#define TCSAFLUSH          3    /* wait for output to finish and flush input */

/* Queue_selector values for tcflush().  POSIX Sec. 7.2.2.2. */
#define TCIFLUSH           1    /* flush accumulated input data */
#define TCOFLUSH           2    /* flush accumulated output data */
```

```
#define TCIOFLUSH          3    /* flush accumulated input and output data */

/* Action values for tcflow().  POSIX Sec. 7.2.2.2. */
#define TCOOFF             1    /* suspend output */
#define TCOON              2    /* restart suspended output */
#define TCIOFF             3    /* transmit a STOP character on the line */
#define TCION              4    /* transmit a START character on the line */


/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( int tcsendbreak, (int _fildes, int _duration)                     );
_PROTOTYPE( int tcdrain, (int _filedes)                                       );
_PROTOTYPE( int tcflush, (int _filedes, int _queue_selector)                  );
_PROTOTYPE( int tcflow, (int _filedes, int _action)                           );
_PROTOTYPE( speed_t cfgetispeed, (const struct termios *_termios_p)           );
_PROTOTYPE( speed_t cfgetospeed, (const struct termios *_termios_p)           );
_PROTOTYPE( int cfsetispeed, (struct termios *_termios_p, speed_t _speed)     );
_PROTOTYPE( int cfsetospeed, (struct termios *_termios_p, speed_t _speed)     );
_PROTOTYPE( int tcgetattr, (int _filedes, struct termios *_termios_p)         );
_PROTOTYPE( int tcsetattr, \
        (int _filedes, int _opt_actions, const struct termios *_termios_p)    );

#define cfgetispeed(termios_p)          ((termios_p)->c_ispeed)
#define cfgetospeed(termios_p)          ((termios_p)->c_ospeed)
#define cfsetispeed(termios_p, speed)   ((termios_p)->c_ispeed = (speed), 0)
#define cfsetospeed(termios_p, speed)   ((termios_p)->c_ospeed = (speed), 0)

#ifdef _MINIX
/* Here are the local extensions to the POSIX standard for Minix. Posix
 * conforming programs are not able to access these, and therefore they are
 * only defined when a Minix program is compiled.
 */

/* Extensions to the termios c_iflag bit map.  */
#define IXANY           0x0800  /* allow any key to continue ouptut */

/* Extensions to the termios c_oflag bit map. They are only active iff
 * OPOST is enabled. */
#define ONLCR           0x0002  /* Map NL to CR-NL on output */
#define XTABS           0x0004  /* Expand tabs to spaces */
#define ONOEOT          0x0008  /* discard EOT's (^D) on output) */

/* Extensions to the termios c_lflag bit map.  */
#define LFLUSHO         0x0200  /* Flush output. */

/* Extensions to the c_cc array. */
#define VREPRINT          11    /* cc_c[VREPRINT] (^R) */
#define VLNEXT            12    /* cc_c[VLNEXT] (^V) */
#define VDISCARD          13    /* cc_c[VDISCARD] (^O) */

/* Extensions to baud rate settings. */
#define B57600          0x0100  /* 57600 baud */
#define B115200         0x0200  /* 115200 baud */

/* These are the default settings used by the kernel and by 'stty sane' */

```

```
#define TCTRL_DEF       (CREAD | CS8 | HUPCL)
#define TINPUT_DEF      (BRKINT | ICRNL | IXON | IXANY)
#define TOUTPUT_DEF     (OPOST | ONLCR)
#define TLOCAL_DEF      (ISIG | IEXTEN | ICANON | ECHO | ECHOE)
#define TSPEED_DEF      B9600

#define TEOF_DEF        '\4'    /* ^D */
#define TEOL_DEF        _POSIX_VDISABLE
#define TERASE_DEF      '\10'   /* ^H */
#define TINTR_DEF       '\177'  /* ^? */
#define TKILL_DEF       '\25'   /* ^U */
#define TMIN_DEF        1
#define TQUIT_DEF       '\34'   /* ^\ */
#define TSTART_DEF      '\21'   /* ^Q */
#define TSTOP_DEF       '\23'   /* ^S */
#define TSUSP_DEF       '\32'   /* ^Z */
#define TTIME_DEF       0
#define TREPRINT_DEF    '\22'   /* ^R */
#define TLNEXT_DEF      '\26'   /* ^V */
#define TDISCARD_DEF    '\17'   /* ^O */

/* Window size. This information is stored in the TTY driver but not used.
 * This can be used for screen based applications in a window environment.
 * The ioctls TIOCGWINSZ and TIOCSWINSZ can be used to get and set this
 * information.
 */

struct winsize
{
        unsigned short  ws_row;         /* rows, in characters */
        unsigned short  ws_col;         /* columns, in characters */
        unsigned short  ws_xpixel;      /* horizontal size, pixels */
        unsigned short  ws_ypixel;      /* vertical size, pixels */
};
#endif /* _MINIX */

#endif /* _TERMIOS_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/a.out.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <a.out> header file describes the format of executable files. */

#ifndef _AOUT_H
#define _AOUT_H

struct  exec {                  /* a.out header */
  unsigned char a_magic[2];     /* magic number */
  unsigned char a_flags;        /* flags, see below */
  unsigned char a_cpu;          /* cpu id */
  unsigned char a_hdrlen;       /* length of header */
  unsigned char a_unused;       /* reserved for future use */
  unsigned short a_version;     /* version stamp (not used at present) */
  long          a_text;         /* size of text segement in bytes */
  long          a_data;         /* size of data segment in bytes */
  long          a_bss;          /* size of bss segment in bytes */
```

```
    long          a_entry;        /* entry point */
    long          a_total;        /* total memory allocated */
    long          a_syms;         /* size of symbol table */

    /* SHORT FORM ENDS HERE */
    long          a_trsize;       /* text relocation size */
    long          a_drsize;       /* data relocation size */
    long          a_tbase;        /* text relocation base */
    long          a_dbase;        /* data relocation base */
  };

  #define A_MAGIC0      (unsigned char) 0x01
  #define A_MAGIC1      (unsigned char) 0x03
  #define BADMAG(X)     ((X).a_magic[0] != A_MAGIC0 ||(X).a_magic[1] != A_MAGIC1)

  /* CPU Id of TARGET machine (byte order coded in low order two bits) */
  #define A_NONE  0x00    /* unknown */
  #define A_I8086 0x04    /* intel i8086/8088 */
  #define A_M68K  0x0B    /* motorola m68000 */
  #define A_NS16K 0x0C    /* national semiconductor 16032 */
  #define A_I80386 0x10   /* intel i80386 */
  #define A_SPARC 0x17    /* Sun SPARC */

  #define A_BLR(cputype)  ((cputype&0x01)!=0) /* TRUE if bytes left-to-right */
  #define A_WLR(cputype)  ((cputype&0x02)!=0) /* TRUE if words left-to-right */

  /* Flags. */
  #define A_UZP   0x01    /* unmapped zero page (pages) */
  #define A_PAL   0x02    /* page aligned executable */
  #define A_NSYM  0x04    /* new style symbol table */
  #define A_EXEC  0x10    /* executable */
  #define A_SEP   0x20    /* separate I/D */
  #define A_PURE  0x40    /* pure text */         /* not used */
  #define A_TOVLY 0x80    /* text overlay */      /* not used */

  /* Offsets of various things. */
  #define A_MINHDR        32
  #define A_TEXTPOS(X)    ((long)(X).a_hdrlen)
  #define A_DATAPOS(X)    (A_TEXTPOS(X) + (X).a_text)
  #define A_HASRELS(X)    ((X).a_hdrlen > (unsigned char) A_MINHDR)
  #define A_HASEXT(X)     ((X).a_hdrlen > (unsigned char) (A_MINHDR +  8))
  #define A_HASLNS(X)     ((X).a_hdrlen > (unsigned char) (A_MINHDR + 16))
  #define A_HASTOFF(X)    ((X).a_hdrlen > (unsigned char) (A_MINHDR + 24))
  #define A_TRELPOS(X)    (A_DATAPOS(X) + (X).a_data)
  #define A_DRELPOS(X)    (A_TRELPOS(X) + (X).a_trsize)
  #define A_SYMPOS(X)     (A_TRELPOS(X) + (A_HASRELS(X) ? \
                          ((X).a_trsize + (X).a_drsize) : 0))

  struct reloc {
    long r_vaddr;                 /* virtual address of reference */
    unsigned short r_symndx;      /* internal segnum or extern symbol num */
    unsigned short r_type;        /* relocation type */
  };

  /* r_tyep values: */
  #define R_ABBS          0
  #define R_RELLBYTE      2
  #define R_PCRBYTE       3
  #define R_RELWORD       4
  #define R_PCRWORD       5
```

```
#define R_RELLONG       6
#define R_PCRLONG       7
#define R_REL3BYTE      8
#define R_KBRANCHE      9

/* r_symndx for internal segments */
#define S_ABS           ((unsigned short)-1)
#define S_TEXT          ((unsigned short)-2)
#define S_DATA          ((unsigned short)-3)
#define S_BSS           ((unsigned short)-4)

struct nlist {                  /* symbol table entry */
  char n_name[8];               /* symbol name */
  long n_value;                 /* value */
  unsigned char n_sclass;       /* storage class */
  unsigned char n_numaux;       /* number of auxiliary entries (not used) */
  unsigned short n_type;        /* language base and derived type (not used) */
};

/* Low bits of storage class (section). */
#define N_SECT            07    /* section mask */
#define N_UNDF            00    /* undefined */
#define N_ABS             01    /* absolute */
#define N_TEXT            02    /* text */
#define N_DATA            03    /* data */
#define N_BSS             04    /* bss */
#define N_COMM            05    /* (common) */

/* High bits of storage class. */
#define N_CLASS         0370    /* storage class mask */
#define C_NULL
#define C_EXT           0020    /* external symbol */
#define C_STAT          0030    /* static */

/* Function prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( int nlist, (char *_file, struct nlist *_nl)                 );

#endif /* _AOUT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/types.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The <sys/types.h> header contains important data type definitions.
 * It is considered good programming practice to use these definitions,
 * instead of the underlying base type.  By convention, all type names end
 * with _t.
 */

#ifndef _TYPES_H
#define _TYPES_H

/* _ANSI is somehow used to determine whether or not the compiler is a
```

```
 * 16 bit compiler
 */
#ifndef _ANSI
#include <ansi.h>
#endif

/* The type size_t holds all results of the sizeof operator.  At first glance,
 * it seems obvious that it should be an unsigned int, but this is not always
 * the case. For example, MINIX-ST (68000) has 32-bit pointers and 16-bit
 * integers. When one asks for the size of a 70K struct or array, the result
 * requires 17 bits to express, so size_t must be a long type.  The type
 * ssize_t is the signed version of size_t.
 */
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned int size_t;
#endif

#ifndef _SSIZE_T
#define _SSIZE_T
typedef int ssize_t;
#endif

#ifndef _TIME_T
#define _TIME_T
typedef long time_t;               /* time in sec since 1 Jan 1970 0000 GMT */
#endif

#ifndef _CLOCK_T
#define _CLOCK_T
typedef long clock_t;              /* unit for system accounting */
#endif

#ifndef _SIGSET_T
#define _SIGSET_T
typedef unsigned long sigset_t;
#endif

/* Types used in disk, inode, etc. data structures. */
typedef short          dev_t;      /* holds (major|minor) device pair */
typedef char           gid_t;      /* group id */
typedef unsigned short ino_t;      /* i-node number */
typedef unsigned short mode_t;     /* file type and permissions bits */
typedef char         nlink_t;      /* number of links to a file */
typedef unsigned long  off_t;      /* offset within a file */
typedef int            pid_t;      /* process id (must be signed) */
typedef short          uid_t;      /* user id */
typedef unsigned long zone_t;      /* zone number */
typedef unsigned long block_t;     /* block number */
typedef unsigned long  bit_t;      /* bit number in a bit map */
typedef unsigned short zone1_t;    /* zone number for V1 file systems */
typedef unsigned short bitchunk_t; /* collection of bits in a bitmap */

typedef unsigned char   u8_t;      /* 8 bit type */
typedef unsigned short u16_t;      /* 16 bit type */
typedef unsigned long  u32_t;      /* 32 bit type */

typedef char            i8_t;      /* 8 bit signed type */
typedef short          i16_t;      /* 16 bit signed type */
typedef long           i32_t;      /* 32 bit signed type */
```

```

/* The following types are needed because MINIX uses K&R style function
 * definitions (for maximum portability).  When a short, such as dev_t, is
 * passed to a function with a K&R definition, the compiler automatically
 * promotes it to an int.  The prototype must contain an int as the parameter,
 * not a short, because an int is what an old-style function definition
 * expects.  Thus using dev_t in a prototype would be incorrect.  It would be
 * sufficient to just use int instead of dev_t in the prototypes, but Dev_t
 * is clearer.
 */
typedef int            Dev_t;
typedef int            Gid_t;
typedef int          Nlink_t;
typedef int            Uid_t;
typedef int             U8_t;
typedef unsigned long  U32_t;
typedef int             I8_t;
typedef int            I16_t;
typedef long            I32_t;

/* ANSI C makes writing down the promotion of unsigned types very messy.  When
 * sizeof(short) == sizeof(int), there is no promotion, so the type stays
 * unsigned.  When the compiler is not ANSI, there is usually no loss of
 * unsignedness, and there are usually no prototypes so the promoted type
 * doesn't matter.  The use of types like Ino_t is an attempt to use ints
 * (which are not promoted) while providing information to the reader.
 */

#ifndef _ANSI_H
#include <ansi.h>
#endif

#if _EM_WSIZE == 2 || !defined(_ANSI)
typedef unsigned int      Ino_t;
typedef unsigned int    Zone1_t;
typedef unsigned int Bitchunk_t;
typedef unsigned int      U16_t;
typedef unsigned int  Mode_t;

#else /* _EM_WSIZE == 4, or _EM_WSIZE undefined, or _ANSI defined */
typedef int               Ino_t;
typedef int             Zone1_t;
typedef int          Bitchunk_t;
typedef int               U16_t;
typedef int           Mode_t;

#endif /* _EM_WSIZE == 2, etc */

/* Signal handler type, e.g. SIG_IGN */
#if defined(_ANSI)
typedef void (*sighandler_t) (int);
#else
typedef void (*sighandler_t)();
#endif

#endif /* _TYPES_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/sys/ioctl.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The ioctl.h header declares device controlling operations. */

#ifndef _IOCTL_H
#define _IOCTL_H

#if _EM_WSIZE >= 4
/* Ioctls have the command encoded in the low-order word, and the size
 * of the parameter in the high-order word. The 3 high bits of the high-
 * order word are used to encode the in/out/void status of the parameter.
 */

#define _IOCPARM_MASK   0x1FFF
#define _IOC_VOID       0x20000000
#define _IOCTYPE_MASK   0xFFFF
#define _IOC_IN         0x40000000
#define _IOC_OUT        0x80000000
#define _IOC_INOUT      (_IOC_IN | _IOC_OUT)

#define _IO(x,y)        ((x << 8) | y | _IOC_VOID)
#define _IOR(x,y,t)     ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
                                _IOC_OUT)
#define _IOW(x,y,t)     ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
                                _IOC_IN)
#define _IORW(x,y,t)    ((x << 8) | y | ((sizeof(t) & _IOCPARM_MASK) << 16) |\
                                _IOC_INOUT)
#else
/* No fancy encoding on a 16-bit machine. */

#define _IO(x,y)        ((x << 8) | y)
#define _IOR(x,y,t)     _IO(x,y)
#define _IOW(x,y,t)     _IO(x,y)
#define _IORW(x,y,t)    _IO(x,y)
#endif


/* Terminal ioctls. */
#define TCGETS          _IOR('T',  8, struct termios) /* tcgetattr */
#define TCSETS          _IOW('T',  9, struct termios) /* tcsetattr, TCSANOW */
#define TCSETSW         _IOW('T', 10, struct termios) /* tcsetattr, TCSADRAIN */
#define TCSETSF         _IOW('T', 11, struct termios) /* tcsetattr, TCSAFLUSH */
#define TCSBRK          _IOW('T', 12, int)            /* tcsendbreak */
#define TCDRAIN         _IO ('T', 13)                 /* tcdrain */
#define TCFLOW          _IOW('T', 14, int)            /* tcflow */
#define TCFLSH          _IOW('T', 15, int)            /* tcflush */
#define TIOCGWINSZ      _IOR('T', 16, struct winsize)
#define TIOCSWINSZ      _IOW('T', 17, struct winsize)
#define TIOCGPGRP       _IOW('T', 18, int)
#define TIOCSPGRP       _IOW('T', 19, int)
#define TIOCSFON        _IOW('T', 20, u8_t [8192])

#define TIOCGETP        _IOR('t',  1, struct sgttyb)
#define TIOCSETP        _IOW('t',  2, struct sgttyb)
#define TIOCGETC        _IOR('t',  3, struct tchars)
#define TIOCSETC        _IOW('t',  4, struct tchars)
```

```

/* Network ioctls. */
#define NWIOSETHOPT     _IOW('n', 16, struct nwio_ethopt)
#define NWIOGETHOPT     _IOR('n', 17, struct nwio_ethopt)
#define NWIOGETHSTAT    _IOR('n', 18, struct nwio_ethstat)

#define NWIOSIPCONF     _IOW('n', 32, struct nwio_ipconf)
#define NWIOGIPCONF     _IOR('n', 33, struct nwio_ipconf)
#define NWIOSIPOPT      _IOW('n', 34, struct nwio_ipopt)
#define NWIOGIPOPT      _IOR('n', 35, struct nwio_ipopt)

#define NWIOIPGROUTE    _IORW('n', 40, struct nwio_route)
#define NWIOIPSROUTE    _IOW ('n', 41, struct nwio_route)
#define NWIOIPDROUTE    _IOW ('n', 42, struct nwio_route)

#define NWIOSTCPCONF    _IOW('n', 48, struct nwio_tcpconf)
#define NWIOGTCPCONF    _IOR('n', 49, struct nwio_tcpconf)
#define NWIOTCPCONN     _IOW('n', 50, struct nwio_tcpcl)
#define NWIOTCPLISTEN   _IOW('n', 51, struct nwio_tcpcl)
#define NWIOTCPATTACH   _IOW('n', 52, struct nwio_tcpatt)
#define NWIOTCPSHUTDOWN _IO ('n', 53)
#define NWIOSTCPOPT     _IOW('n', 54, struct nwio_tcpopt)
#define NWIOGTCPOPT     _IOR('n', 55, struct nwio_tcpopt)

#define NWIOSUDPOPT     _IOW('n', 64, struct nwio_udpopt)
#define NWIOGUDPOPT     _IOR('n', 65, struct nwio_udpopt)

/* Disk ioctls. */
#define DIOCEJECT       _IO ('d', 5)
#define DIOCSETP        _IOW('d', 6, struct partition)
#define DIOCGETP        _IOR('d', 7, struct partition)

/* Keyboard ioctls. */
#define KIOCSMAP        _IOW('k', 3, keymap_t)

/* Memory ioctls. */
#define MIOCRAMSIZE     _IOW('m', 3, u32_t)     /* Size of the ramdisk */
#define MIOCSPSINFO     _IOW('m', 4, void *)
#define MIOCGPSINFO     _IOR('m', 5, struct psinfo)

/* Magnetic tape ioctls. */
#define MTIOCTOP        _IOW('M', 1, struct mtop)
#define MTIOCGET        _IOR('M', 2, struct mtget)

/* SCSI command. */
#define SCIOCCMD        _IOW('S', 1, struct scsicmd)

/* CD-ROM ioctls. */
#define CDIOPLAYTI      _IOR('c', 1, struct cd_play_track)
#define CDIOPLAYMSS     _IOR('c', 2, struct cd_play_mss)
#define CDIOREADTOCHDR  _IOW('c', 3, struct cd_toc_entry)
#define CDIOREADTOC     _IOW('c', 4, struct cd_toc_entry)
#define CDIOREADSUBCH   _IOW('c', 5, struct cd_toc_entry)
#define CDIOSTOP        _IO ('c', 10)
#define CDIOPAUSE       _IO ('c', 11)
#define CDIORESUME      _IO ('c', 12)
#define CDIOEJECT       DIOCEJECT

/* Soundcard DSP ioctls. */
#define DSPIORATE       _IOR('s', 1, unsigned int)
```

```
#define DSPIOSTEREO     _IOR('s', 2, unsigned int)
#define DSPIOSIZE       _IOR('s', 3, unsigned int)
#define DSPIOBITS       _IOR('s', 4, unsigned int)
#define DSPIOSIGN       _IOR('s', 5, unsigned int)
#define DSPIOMAX        _IOW('s', 6, unsigned int)
#define DSPIORESET      _IO ('s', 7)

/* Soundcard mixer ioctls. */
#define MIXIOGETVOLUME          _IORW('s', 10, struct volume_level)
#define MIXIOGETINPUTLEFT       _IORW('s', 11, struct inout_ctrl)
#define MIXIOGETINPUTRIGHT      _IORW('s', 12, struct inout_ctrl)
#define MIXIOGETOUTPUT          _IORW('s', 13, struct inout_ctrl)
#define MIXIOSETVOLUME          _IORW('s', 20, struct volume_level)
#define MIXIOSETINPUTLEFT       _IORW('s', 21, struct inout_ctrl)
#define MIXIOSETINPUTRIGHT      _IORW('s', 22, struct inout_ctrl)
#define MIXIOSETOUTPUT          _IORW('s', 23, struct inout_ctrl)

#ifndef _ANSI
#include <ansi.h>
#endif

_PROTOTYPE( int ioctl, (int _fd, int _request, void *_data)            );

#endif /* _IOCTL_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/sigcontext.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _SIGCONTEXT_H
#define _SIGCONTEXT_H

/* The sigcontext structure is used by the sigreturn(2) system call.
 * sigreturn() is seldom called by user programs, but it is used internally
 * by the signal catching mechanism.
 */

#ifndef _ANSI_H
#include <ansi.h>
#endif

#ifndef _CONFIG_H
#include <minix/config.h>
#endif

#if !defined(CHIP)
#include "error, configuration is not known"
#endif

/* The following structure should match the stackframe_s structure used
 * by the kernel's context switching code.  Floating point registers should
 * be added in a different struct.
 */
#if (CHIP == INTEL)
struct sigregs {
#if _WORD_SIZE == 4
  short sr_gs;
  short sr_fs;
#endif /* _WORD_SIZE == 4 */
```

```
  short sr_es;
  short sr_ds;
  int sr_di;
  int sr_si;
  int sr_bp;
  int sr_st;                  /* stack top -- used in kernel */
  int sr_bx;
  int sr_dx;
  int sr_cx;
  int sr_retreg;
  int sr_retadr;              /* return address to caller of save -- used
                               * in kernel */
  int sr_pc;
  int sr_cs;
  int sr_psw;
  int sr_sp;
  int sr_ss;
};

struct sigframe {             /* stack frame created for signalled process */
  _PROTOTYPE( void (*sf_retadr), (void) );
  int sf_signo;
  int sf_code;
  struct sigcontext *sf_scp;
  int sf_fp;
  _PROTOTYPE( void (*sf_retadr2), (void) );
  struct sigcontext *sf_scpcopy;
};

#else
#if (CHIP == M68000)
struct sigregs {
  long sr_retreg;                     /* d0 */
  long sr_d1;
  long sr_d2;
  long sr_d3;
  long sr_d4;
  long sr_d5;
  long sr_d6;
  long sr_d7;
  long sr_a0;
  long sr_a1;
  long sr_a2;
  long sr_a3;
  long sr_a4;
  long sr_a5;
  long sr_a6;
  long sr_sp;                 /* also known as a7 */
  long sr_pc;
  short sr_psw;
  short sr_dummy;             /* make size multiple of 4 for system.c */
};
#else
#include "error, CHIP is not supported"
#endif
#endif /* CHIP == INTEL */

struct sigcontext {
  int sc_flags;               /* sigstack state to restore */
  long sc_mask;               /* signal mask to restore */
```

```
  struct sigregs sc_regs;       /* register set to restore */
};

#if (CHIP == INTEL)
#if _WORD_SIZE == 4
#define sc_gs sc_regs.sr_gs
#define sc_fs sc_regs.sr_fs
#endif /* _WORD_SIZE == 4 */
#define sc_es sc_regs.sr_es
#define sc_ds sc_regs.sr_ds
#define sc_di sc_regs.sr_di
#define sc_si sc_regs.sr_si
#define sc_fp sc_regs.sr_bp
#define sc_st sc_regs.sr_st             /* stack top -- used in kernel */
#define sc_bx sc_regs.sr_bx
#define sc_dx sc_regs.sr_dx
#define sc_cx sc_regs.sr_cx
#define sc_retreg sc_regs.sr_retreg
#define sc_retadr sc_regs.sr_retadr     /* return address to caller of
                                           save -- used in kernel */
#define sc_pc sc_regs.sr_pc
#define sc_cs sc_regs.sr_cs
#define sc_psw sc_regs.sr_psw
#define sc_sp sc_regs.sr_sp
#define sc_ss sc_regs.sr_ss
#endif /* CHIP == INTEL */

#if (CHIP == M68000)
#define sc_retreg sc_regs.sr_retreg
#define sc_d1 sc_regs.sr_d1
#define sc_d2 sc_regs.sr_d2
#define sc_d3 sc_regs.sr_d3
#define sc_d4 sc_regs.sr_d4
#define sc_d5 sc_regs.sr_d5
#define sc_d6 sc_regs.sr_d6
#define sc_d7 sc_regs.sr_d7
#define sc_a0 sc_regs.sr_a0
#define sc_a1 sc_regs.sr_a1
#define sc_a2 sc_regs.sr_a2
#define sc_a3 sc_regs.sr_a3
#define sc_a4 sc_regs.sr_a4
#define sc_a5 sc_regs.sr_a5
#define sc_fp sc_regs.sr_a6
#define sc_sp sc_regs.sr_sp
#define sc_pc sc_regs.sr_pc
#define sc_psw sc_regs.sr_psw
#endif /* CHIP == M68000 */

/* Values for sc_flags.  Must agree with <minix/jmp_buf.h>. */
#define SC_SIGCONTEXT   2       /* nonzero when signal context is included */
#define SC_NOREGLOCALS  4       /* nonzero when registers are not to be
                                           saved and restored */

_PROTOTYPE( int sigreturn, (struct sigcontext *_scp)                    );

#endif /* _SIGCONTEXT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/ptrace.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* <sys/ptrace.h>
 * definitions for ptrace(2)
 */

#ifndef _PTRACE_H
#define _PTRACE_H

#define T_STOP         -1       /* stop the process */
#define T_OK            0       /* enable tracing by parent for this process */
#define T_GETINS        1       /* return value from instruction space */
#define T_GETDATA       2       /* return value from data space */
#define T_GETUSER       3       /* return value from user process table */
#define T_SETINS        4       /* set value from instruction space */
#define T_SETDATA       5       /* set value from data space */
#define T_SETUSER       6       /* set value in user process table */
#define T_RESUME        7       /* resume execution */
#define T_EXIT          8       /* exit */
#define T_STEP          9       /* set trace bit */

/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( long ptrace, (int _req, pid_t _pid, long _addr, long _data) );

#endif /* _PTRACE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/stat.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The <sys/stat.h> header defines a struct that is used in the stat() and
 * fstat functions.  The information in this struct comes from the i-node of
 * some file.  These calls are the only approved way to inspect i-nodes.
 */

#ifndef _STAT_H
#define _STAT_H

struct stat {
  dev_t st_dev;                 /* major/minor device number */
  ino_t st_ino;                 /* i-node number */
  mode_t st_mode;               /* file mode, protection bits, etc. */
  short int st_nlink;           /* # links; TEMPORARY HACK: should be nlink_t*/
  uid_t st_uid;                 /* uid of the file's owner */
  short int st_gid;             /* gid; TEMPORARY HACK: should be gid_t */
  dev_t st_rdev;
  off_t st_size;                /* file size */
  time_t st_atime;              /* time of last access */
  time_t st_mtime;              /* time of last data modification */
  time_t st_ctime;              /* time of last file status change */
```

```
};

/* Traditional mask definitions for st_mode. */
/* The ugly casts on only some of the definitions are to avoid suprising sign
 * extensions such as S_IFREG != (mode_t) S_IFREG when ints are 32 bits.
 */
#define S_IFMT  ((mode_t) 0170000)      /* type of file */
#define S_IFREG ((mode_t) 0100000)      /* regular */
#define S_IFBLK 0060000         /* block special */
#define S_IFDIR 0040000         /* directory */
#define S_IFCHR 0020000         /* character special */
#define S_IFIFO 0010000         /* this is a FIFO */
#define S_ISUID 0004000         /* set user id on execution */
#define S_ISGID 0002000         /* set group id on execution */
                                /* next is reserved for future use */
#define S_ISVTX   01000         /* save swapped text even after use */

/* POSIX masks for st_mode. */
#define S_IRWXU   00700         /* owner:  rwx------ */
#define S_IRUSR   00400         /* owner:  r-------- */
#define S_IWUSR   00200         /* owner:  -w------- */
#define S_IXUSR   00100         /* owner:  --x------ */

#define S_IRWXG   00070         /* group:  ---rwx--- */
#define S_IRGRP   00040         /* group:  ---r----- */
#define S_IWGRP   00020         /* group:  ----w---- */
#define S_IXGRP   00010         /* group:  -----x--- */

#define S_IRWXO   00007         /* others: ------rwx */
#define S_IROTH   00004         /* others: ------r-- */
#define S_IWOTH   00002         /* others: -------w- */
#define S_IXOTH   00001         /* others: --------x */

/* The following macros test st_mode (from POSIX Sec. 5.6.1.1). */
#define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)     /* is a reg file */
#define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)     /* is a directory */
#define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)     /* is a char spec */
#define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)     /* is a block spec */
#define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)     /* is a pipe/FIFO */


/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( int chmod, (const char *_path, Mode_t _mode)                );
_PROTOTYPE( int fstat, (int _fildes, struct stat *_buf)                 );
_PROTOTYPE( int mkdir, (const char *_path, Mode_t _mode)                );
_PROTOTYPE( int mkfifo, (const char *_path, Mode_t _mode)               );
_PROTOTYPE( int stat, (const char *_path, struct stat *_buf)            );
_PROTOTYPE( mode_t umask, (Mode_t _cmask)                               );

#endif /* _STAT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/dir.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The <dir.h> header gives the layout of a directory. */

#ifndef _DIR_H
#define _DIR_H

#define DIRBLKSIZ       512     /* size of directory block */

#ifndef DIRSIZ
#define DIRSIZ  14
#endif

struct direct {
  ino_t d_ino;
  char d_name[DIRSIZ];
};

#endif /* _DIR_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/sys/wait.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The <sys/wait.h> header contains macros related to wait(). The value
 * returned by wait() and waitpid() depends on whether the process
 * terminated by an exit() call, was killed by a signal, or was stopped
 * due to job control, as follows:
 *
 *                               High byte   Low byte
 *                              +---------------------+
 *      exit(status)            |  status  |    0     |
 *                              +---------------------+
 *      killed by signal        |    0     |  signal  |
 *                              +---------------------+
 *      stopped (job control)   |  signal  |   0177   |
 *                              +---------------------+
 */

#ifndef _WAIT_H
#define _WAIT_H

#define _LOW(v)         ( (v) & 0377)
#define _HIGH(v)        ( ((v) >> 8) & 0377)

#define WNOHANG         1       /* do not wait for child to exit */
#define WUNTRACED       2       /* for job control; not implemented */

#define WIFEXITED(s)    (_LOW(s) == 0)                      /* normal exit */
#define WEXITSTATUS(s)  (_HIGH(s))                          /* exit status */
#define WTERMSIG(s)     (_LOW(s) & 0177)                    /* sig value */
#define WIFSIGNALED(s)  (((unsigned int)(s)-1 & 0xFFFF) < 0xFF) /* signaled */
#define WIFSTOPPED(s)   (_LOW(s) == 0177)                   /* stopped */
#define WSTOPSIG(s)     (_HIGH(s) & 0377)                   /* stop signal */
```

```

/* Function Prototypes. */
#ifndef _ANSI_H
#include <ansi.h>
#endif

_PROTOTYPE( pid_t wait, (int *_stat_loc)                                   );
_PROTOTYPE( pid_t waitpid, (pid_t _pid, int *_stat_loc, int _options)      );

#endif /* _WAIT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/minix/config.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef _CONFIG_H
#define _CONFIG_H

/* Minix release and version numbers. */
#define OS_RELEASE "2.0"
#define OS_VERSION "0"

/* This file sets configuration parameters for the MINIX kernel, FS, and MM.
 * It is divided up into two main sections.  The first section contains
 * user-settable parameters.  In the second section, various internal system
 * parameters are set based on the user-settable parameters.
 */

/*===========================================================================*
 *              This section contains user-settable parameters               *
 *===========================================================================*/
#define MACHINE       IBM_PC    /* Must be one of the names listed below */

#define IBM_PC             1    /* any  8088 or 80x86-based system */
#define SUN_4             40    /* any Sun SPARC-based system */
#define SUN_4_60          40    /* Sun-4/60 (aka SparcStation 1 or Campus) */
#define ATARI             60    /* ATARI ST/STe/TT (68000/68030) */
#define AMIGA             61    /* Commodore Amiga (68000) */
#define MACINTOSH         62    /* Apple Macintosh (68000) */

/* Word size in bytes (a constant equal to sizeof(int)). */
#if __ACK__
#define _WORD_SIZE      _EM_WSIZE
#endif


/* If ROBUST is set to 1, writes of i-node, directory, and indirect blocks
 * from the cache happen as soon as the blocks are modified.  This gives a more
 * robust, but slower, file system.  If it is set to 0, these blocks are not
 * given any special treatment, which may cause problems if the system crashes.
 */
#define ROBUST             0    /* 0 for speed, 1 for robustness */

/* Number of slots in the process table for user processes. */
#define NR_PROCS          32
```

```

/* The buffer cache should be made as large as you can afford. */
#if (MACHINE == IBM_PC && _WORD_SIZE == 2)
#define NR_BUFS           40    /* # blocks in the buffer cache */
#define NR_BUF_HASH       64    /* size of buf hash table; MUST BE POWER OF 2*/
#endif

#if (MACHINE == IBM_PC && _WORD_SIZE == 4)
#define NR_BUFS          512    /* # blocks in the buffer cache */
#define NR_BUF_HASH     1024    /* size of buf hash table; MUST BE POWER OF 2*/
#endif

#if (MACHINE == SUN_4_60)
#define NR_BUFS          512    /* # blocks in the buffer cache (<=1536) */
#define NR_BUF_HASH      512    /* size of buf hash table; MUST BE POWER OF 2*/
#endif

#if (MACHINE == ATARI)
#define NR_BUFS         1536    /* # blocks in the buffer cache (<=1536) */
#define NR_BUF_HASH     2048    /* size of buf hash table; MUST BE POWER OF 2*/
#endif

/* Defines for kernel configuration. */
#define AUTO_BIOS          0    /* xt_wini.c - use Western's autoconfig BIOS */
#define LINEWRAP           1    /* console.c - wrap lines at column 80 */
#define ALLOW_GAP_MESSAGES 1    /* proc.c - allow messages in the gap between
                                 * the end of bss and lowest stack address */

/* Enable or disable the second level file system cache on the RAM disk. */
#define ENABLE_CACHE2      0

/* Include or exclude device drivers.  Set to 1 to include, 0 to exclude. */
#define ENABLE_NETWORKING  0    /* enable TCP/IP code */
#define ENABLE_AT_WINI     1    /* enable AT winchester driver */
#define ENABLE_BIOS_WINI   0    /* enable BIOS winchester driver */
#define ENABLE_ESDI_WINI   0    /* enable ESDI winchester driver */
#define ENABLE_XT_WINI     0    /* enable XT winchester driver */
#define ENABLE_ADAPTEC_SCSI 0   /* enable ADAPTEC SCSI driver */
#define ENABLE_MITSUMI_CDROM 0  /* enable Mitsumi CD-ROM driver */
#define ENABLE_SB_AUDIO    0    /* enable Soundblaster audio driver */

/* DMA_SECTORS may be increased to speed up DMA based drivers. */
#define DMA_SECTORS        1    /* DMA buffer size (must be >= 1) */

/* Include or exclude backwards compatibility code. */
#define ENABLE_BINCOMPAT   0    /* for binaries using obsolete calls */
#define ENABLE_SRCCOMPAT   0    /* for sources using obsolete calls */

/* Determine which device to use for pipes. */
#define PIPE_DEV    ROOT_DEV    /* put pipes on root device */

/* NR_CONS, NR_RS_LINES, and NR_PTYS determine the number of terminals the
 * system can handle.
 */
#define NR_CONS            2    /* # system consoles (1 to 8) */
#define NR_RS_LINES        0    /* # rs232 terminals (0, 1, or 2) */
#define NR_PTYS            0    /* # pseudo terminals (0 to 64) */

#if (MACHINE == ATARI)
/* The next define says if you have an ATARI ST or TT */
```

```
#define ATARI_TYPE        TT
#define ST                 1    /* all ST's and Mega ST's */
#define STE                2    /* all STe and Mega STe's */
#define TT                 3

/* if SCREEN is set to 1 graphical screen operations are possible */
#define SCREEN             1

/* This define says whether the keyboard generates VT100 or IBM_PC escapes. */
#define KEYBOARD       VT100    /* either VT100 or IBM_PC */
#define VT100            100

/* The next define determines the kind of partitioning. */
#define PARTITIONING   SUPRA    /* one of the following or ATARI */
#define SUPRA              1    /*ICD, SUPRA and BMS are all the same */
#define BMS                1
#define ICD                1
#define CBHD               2
#define EICKMANN           3

/* Define the number of hard disk drives on your system. */
#define NR_ACSI_DRIVES     3    /* typically 0 or 1 */
#define NR_SCSI_DRIVES     1    /* typically 0 (ST, STe) or 1 (TT) */

/* Some systems need to have a little delay after each winchester
 * commands. These systems need FAST_DISK set to 0. Other disks do not
 * need this delay, and thus can have FAST_DISK set to 1 to avoid this delay.
 */
#define FAST_DISK          1    /* 0 or 1 */

/* Note: if you want to make your kernel smaller, you can set NR_FD_DRIVES
 * to 0. You will still be able to boot minix.img from floppy. However, you
 * MUST fetch both the root and usr filesystem from a hard disk
 */

/* Define the number of floppy disk drives on your system. */
#define NR_FD_DRIVES       1    /* 0, 1, 2 */

/* This configuration define controls parallel printer code. */
#define PAR_PRINTER        1    /* disable (0) / enable (1) parallel printer */

/* This configuration define controls disk controller clock code. */
#define HD_CLOCK           1    /* disable (0) / enable (1) hard disk clock */

#endif


/*=========================================================================*
 *      There are no user-settable parameters after this line              *
 *=========================================================================*/
/* Set the CHIP type based on the machine selected. The symbol CHIP is actually
 * indicative of more than just the CPU.  For example, machines for which
 * CHIP == INTEL are expected to have 8259A interrrupt controllers and the
 * other properties of IBM PC/XT/AT/386 types machines in general. */
#define INTEL             1     /* CHIP type for PC, XT, AT, 386 and clones */
#define M68000            2     /* CHIP type for Atari, Amiga, Macintosh    */
#define SPARC             3     /* CHIP type for SUN-4 (e.g. SPARCstation)  */

/* Set the FP_FORMAT type based on the machine selected, either hw or sw    */
#define FP_NONE           0     /* no floating point support                */
```

```
#define FP_IEEE           1     /* conform IEEE floating point standard     */

#if (MACHINE == IBM_PC)
#define CHIP           INTEL
#define SHADOWING         0
#define ENABLE_WINI      (ENABLE_AT_WINI || ENABLE_BIOS_WINI || \
                          ENABLE_ESDI_WINI || ENABLE_XT_WINI)
#define ENABLE_SCSI      (ENABLE_ADAPTEC_SCSI)
#define ENABLE_CDROM     (ENABLE_MITSUMI_CDROM)
#define ENABLE_AUDIO     (ENABLE_SB_AUDIO)
#endif

#if (MACHINE == ATARI) || (MACHINE == AMIGA) || (MACHINE == MACINTOSH)
#define CHIP          M68000
#define SHADOWING         1
#endif

#if (MACHINE == SUN_4) || (MACHINE == SUN_4_60)
#define CHIP           SPARC
#define FP_FORMAT   FP_IEEE
#define SHADOWING         0
#endif

#if (MACHINE == ATARI) || (MACHINE == SUN_4)
#define ASKDEV            1     /* ask for boot device */
#define FASTLOAD          1     /* use multiple block transfers to init ram */
#endif

#if (ATARI_TYPE == TT) /* and all other 68030's */
#define FPP
#undef SHADOWING
#define SHADOWING 0
#endif

#ifndef FP_FORMAT
#define FP_FORMAT   FP_NONE
#endif

/* The file buf.h uses MAYBE_WRITE_IMMED. */
#if ROBUST
#define MAYBE_WRITE_IMMED  WRITE_IMMED  /* slower but perhaps safer */
#else
#define MAYBE_WRITE_IMMED 0             /* faster */
#endif

#ifndef MACHINE
error "In <minix/config.h> please define MACHINE"
#endif

#ifndef CHIP
error "In <minix/config.h> please define MACHINE to have a legal value"
#endif

#if (MACHINE == 0)
error "MACHINE has incorrect value (0)"
#endif

#endif /* _CONFIG_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Copyright (C) 1997 by Prentice-Hall, Inc.  Permission is hereby granted
 * to redistribute the binary and source programs of this system for
 * educational or research purposes.  For other use, written permission from
 * Prentice-Hall is required.
 */

#define EXTERN        extern   /* used in *.h files */
#define PRIVATE       static   /* PRIVATE x limits the scope of x */
#define PUBLIC                 /* PUBLIC is the opposite of PRIVATE */
#define FORWARD       static   /* some compilers require this to be 'static'*/

#define TRUE               1   /* used for turning integers into Booleans */
#define FALSE              0   /* used for turning integers into Booleans */

#define HZ                60   /* clock freq (software settable on IBM-PC) */
#define BLOCK_SIZE      1024   /* # bytes in a disk block */
#define SUPER_USER (uid_t) 0   /* uid_t of superuser */

#define MAJOR              8   /* major device = (dev>>MAJOR) & 0377 */
#define MINOR              0   /* minor device = (dev>>MINOR) & 0377 */

#define NULL     ((void *)0)   /* null pointer */
#define CPVEC_NR          16   /* max # of entries in a SYS_VCOPY request */
#define NR_IOREQS       MIN(NR_BUFS, 64)
                               /* maximum number of entries in an iorequest */

#define NR_SEGS            3   /* # segments per process */
#define T                  0   /* proc[i].mem_map[T] is for text */
#define D                  1   /* proc[i].mem_map[D] is for data */
#define S                  2   /* proc[i].mem_map[S] is for stack */

/* Process numbers of some important processes. */
#define MM_PROC_NR         0   /* process number of memory manager */
#define FS_PROC_NR         1   /* process number of file system */
#define INET_PROC_NR       2   /* process number of the TCP/IP server */
#define INIT_PROC_NR    (INET_PROC_NR + ENABLE_NETWORKING)
                               /* init -- the process that goes multiuser */
#define LOW_USER        (INET_PROC_NR + ENABLE_NETWORKING)
                               /* first user not part of operating system */

/* Miscellaneous */
#define BYTE            0377   /* mask for 8 bits */
#define READING            0   /* copy data to user */
#define WRITING            1   /* copy data from user */
#define NO_NUM        0x8000   /* used as numerical argument to panic() */
#define NIL_PTR   (char *) 0   /* generally useful expression */
#define HAVE_SCATTERED_IO  1   /* scattered I/O is now standard */

/* Macros. */
#define MAX(a, b)   ((a) > (b) ? (a) : (b))
#define MIN(a, b)   ((a) < (b) ? (a) : (b))

/* Number of tasks. */
#define NR_TASKS        (9 + ENABLE_WINI + ENABLE_SCSI + ENABLE_CDROM \
                        + ENABLE_NETWORKING + 2 * ENABLE_AUDIO)
```

```

/* Memory is allocated in clicks. */
#if (CHIP == INTEL)
#define CLICK_SIZE       256    /* unit in which memory is allocated */
#define CLICK_SHIFT        8    /* log2 of CLICK_SIZE */
#endif

#if (CHIP == SPARC) || (CHIP == M68000)
#define CLICK_SIZE      4096    /* unit in which memory is alocated */
#define CLICK_SHIFT       12    /* 2log of CLICK_SIZE */
#endif

#define click_to_round_k(n) \
        ((unsigned) ((((unsigned long) (n) << CLICK_SHIFT) + 512) / 1024))
#if CLICK_SIZE < 1024
#define k_to_click(n) ((n) * (1024 / CLICK_SIZE))
#else
#define k_to_click(n) ((n) / (CLICK_SIZE / 1024))
#endif

#define ABS             -999    /* this process means absolute memory */

/* Flag bits for i_mode in the inode. */
#define I_TYPE          0170000 /* this field gives inode type */
#define I_REGULAR       0100000 /* regular file, not dir or special */
#define I_BLOCK_SPECIAL 0060000 /* block special file */
#define I_DIRECTORY     0040000 /* file is a directory */
#define I_CHAR_SPECIAL  0020000 /* character special file */
#define I_NAMED_PIPE    0010000 /* named pipe (FIFO) */
#define I_SET_UID_BIT   0004000 /* set effective uid_t on exec */
#define I_SET_GID_BIT   0002000 /* set effective gid_t on exec */
#define ALL_MODES       0006777 /* all bits for user, group and others */
#define RWX_MODES       0000777 /* mode bits for RWX only */
#define R_BIT           0000004 /* Rwx protection bit */
#define W_BIT           0000002 /* rWx protection bit */
#define X_BIT           0000001 /* rwX protection bit */
#define I_NOT_ALLOC     0000000 /* this inode is free */

/* Some limits. */
#define MAX_BLOCK_NR  ((block_t) 077777777)     /* largest block number */
#define HIGHEST_ZONE   ((zone_t) 077777777)     /* largest zone number */
#define MAX_INODE_NR      ((ino_t) 0177777)     /* largest inode number */
#define MAX_FILE_POS ((off_t) 037777777777)     /* largest legal file offset */

#define NO_BLOCK              ((block_t) 0)     /* absence of a block number */
#define NO_ENTRY                ((ino_t) 0)     /* absence of a dir entry */
#define NO_ZONE                ((zone_t) 0)     /* absence of a zone number */
#define NO_DEV                  ((dev_t) 0)     /* absence of a device numb */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  include/minix/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#ifndef _TYPE_H
#define _TYPE_H
#ifndef _MINIX_TYPE_H
#define _MINIX_TYPE_H

/* Type definitions. */
typedef unsigned int vir_clicks; /* virtual  addresses and lengths in clicks */
typedef unsigned long phys_bytes;/* physical addresses and lengths in bytes */
typedef unsigned int phys_clicks;/* physical addresses and lengths in clicks */

#if (CHIP == INTEL)
typedef unsigned int vir_bytes; /* virtual addresses and lengths in bytes */
#endif

#if (CHIP == M68000)
typedef unsigned long vir_bytes;/* virtual addresses and lengths in bytes */
#endif

#if (CHIP == SPARC)
typedef unsigned long vir_bytes;/* virtual addresses and lengths in bytes */
#endif

/* Types relating to messages. */
#define M1                 1
#define M3                 3
#define M4                 4
#define M3_STRING         14

typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
typedef struct {int m2i1, m2i2, m2i3; long m2l1, m2l2; char *m2p1;} mess_2;
typedef struct {int m3i1, m3i2; char *m3p1; char m3ca1[M3_STRING];} mess_3;
typedef struct {long m4l1, m4l2, m4l3, m4l4, m4l5;} mess_4;
typedef struct {char m5c1, m5c2; int m5i1, m5i2; long m5l1, m5l2, m5l3;}mess_5;
typedef struct {int m6i1, m6i2, m6i3; long m6l1; sighandler_t m6f1;} mess_6;

typedef struct {
  int m_source;                 /* who sent the message */
  int m_type;                   /* what kind of message is it */
  union {
        mess_1 m_m1;
        mess_2 m_m2;
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_6 m_m6;
  } m_u;
} message;

/* The following defines provide names for useful members. */
#define m1_i1  m_u.m_m1.m1i1
#define m1_i2  m_u.m_m1.m1i2
#define m1_i3  m_u.m_m1.m1i3
#define m1_p1  m_u.m_m1.m1p1
#define m1_p2  m_u.m_m1.m1p2
#define m1_p3  m_u.m_m1.m1p3
```

```

#define m2_i1  m_u.m_m2.m2i1
#define m2_i2  m_u.m_m2.m2i2
#define m2_i3  m_u.m_m2.m2i3
#define m2_l1  m_u.m_m2.m2l1
#define m2_l2  m_u.m_m2.m2l2
#define m2_p1  m_u.m_m2.m2p1

#define m3_i1  m_u.m_m3.m3i1
#define m3_i2  m_u.m_m3.m3i2
#define m3_p1  m_u.m_m3.m3p1
#define m3_ca1 m_u.m_m3.m3ca1

#define m4_l1  m_u.m_m4.m4l1
#define m4_l2  m_u.m_m4.m4l2
#define m4_l3  m_u.m_m4.m4l3
#define m4_l4  m_u.m_m4.m4l4
#define m4_l5  m_u.m_m4.m4l5

#define m5_c1  m_u.m_m5.m5c1
#define m5_c2  m_u.m_m5.m5c2
#define m5_i1  m_u.m_m5.m5i1
#define m5_i2  m_u.m_m5.m5i2
#define m5_l1  m_u.m_m5.m5l1
#define m5_l2  m_u.m_m5.m5l2
#define m5_l3  m_u.m_m5.m5l3

#define m6_i1  m_u.m_m6.m6i1
#define m6_i2  m_u.m_m6.m6i2
#define m6_i3  m_u.m_m6.m6i3
#define m6_l1  m_u.m_m6.m6l1
#define m6_f1  m_u.m_m6.m6f1

struct mem_map {
  vir_clicks mem_vir;           /* virtual address */
  phys_clicks mem_phys;         /* physical address */
  vir_clicks mem_len;           /* length */
};

struct iorequest_s {
  long io_position;             /* position in device file (really off_t) */
  char *io_buf;                 /* buffer in user space */
  int io_nbytes;                /* size of request */
  unsigned short io_request;    /* read, write (optionally) */
};
#endif /* _TYPE_H */

typedef struct {
  vir_bytes iov_addr;           /* address of an I/O buffer */
  vir_bytes iov_size;           /* sizeof an I/O buffer */
} iovec_t;

typedef struct {
  vir_bytes cpv_src;            /* src address of data */
  vir_bytes cpv_dst;            /* dst address of data */
  vir_bytes cpv_size;           /* size of data */
} cpvec_t;

/* MM passes the address of a structure of this type to KERNEL when
 * do_sendsig() is invoked as part of the signal catching mechanism.
```

```
 * The structure contain all the information that KERNEL needs to build
 * the signal stack.
 */
struct sigmsg {
  int sm_signo;                 /* signal number being caught */
  unsigned long sm_mask;        /* mask to restore when handler returns */
  vir_bytes sm_sighandler;      /* address of handler */
  vir_bytes sm_sigreturn;       /* address of _sigreturn in C library */
  vir_bytes sm_stkptr;          /* user stack pointer */
};

#define MESS_SIZE (sizeof(message))     /* might need usizeof from fs here */
#define NIL_MESS ((message *) 0)

struct psinfo {         /* information for the ps(1) program */
  u16_t nr_tasks, nr_procs;     /* NR_TASKS and NR_PROCS constants. */
  vir_bytes proc, mproc, fproc; /* addresses of the main process tables. */
};

#endif /* _MINIX_TYPE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/minix/syslib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Prototypes for system library functions. */

#ifndef _SYSLIB_H
#define _SYSLIB_H

/* Hide names to avoid name space pollution. */
#define sendrec         _sendrec
#define receive         _receive
#define send            _send

/* Minix user+system library. */
_PROTOTYPE( void printk, (char *_fmt, ...)                              );
_PROTOTYPE( int sendrec, (int _src_dest, message *_m_ptr)               );
_PROTOTYPE( int _taskcall, (int _who, int _syscallnr, message *_msgptr) );

/* Minix system library. */
_PROTOTYPE( int receive, (int _src, message *_m_ptr)                    );
_PROTOTYPE( int send, (int _dest, message *_m_ptr)                      );

_PROTOTYPE( int sys_abort, (int _how, ...)                              );
_PROTOTYPE( int sys_adjmap, (int _proc, struct mem_map *_ptr,
                vir_clicks _data_clicks, vir_clicks _sp)                );
_PROTOTYPE( int sys_copy, (int _src_proc, int _src_seg, phys_bytes _src_vir,
        int _dst_proc, int _dst_seg, phys_bytes _dst_vir, phys_bytes _bytes));
_PROTOTYPE( int sys_exec, (int _proc, char *_ptr, int _traced,
                                char *_aout, vir_bytes _initpc)         );
_PROTOTYPE( int sys_execmap, (int _proc, struct mem_map *_ptr)          );
_PROTOTYPE( int sys_fork, (int _parent, int _child, int _pid,
                                        phys_clicks _shadow)            );
_PROTOTYPE( int sys_fresh, (int _proc, struct mem_map *_ptr,
```

```
        phys_clicks _dc, phys_clicks *_basep, phys_clicks *_sizep)      );
_PROTOTYPE( int sys_getsp, (int _proc, vir_bytes *_newsp)               );
_PROTOTYPE( int sys_newmap, (int _proc, struct mem_map *_ptr)           );
_PROTOTYPE( int sys_getmap, (int _proc, struct mem_map *_ptr)           );
_PROTOTYPE( int sys_sendsig, (int _proc, struct sigmsg *_ptr)           );
_PROTOTYPE( int sys_oldsig, (int _proc, int _sig, sighandler_t _sighandler));
_PROTOTYPE( int sys_endsig, (int _proc)                                 );
_PROTOTYPE( int sys_sigreturn, (int _proc, vir_bytes _scp, int _flags)  );
_PROTOTYPE( int sys_trace, (int _req, int _procnr, long _addr, long *_data_p));
_PROTOTYPE( int sys_xit, (int _parent, int _proc, phys_clicks *_basep,
                                          phys_clicks *_sizep));
_PROTOTYPE( int sys_kill, (int _proc, int _sig)                         );
_PROTOTYPE( int sys_times, (int _proc, clock_t _ptr[5])                 );

#endif /* _SYSLIB_H */
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                include/minix/callnr.h
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#define NCALLS          77    /* number of system calls allowed */

#define EXIT             1
#define FORK             2
#define READ             3
#define WRITE            4
#define OPEN             5
#define CLOSE            6
#define WAIT             7
#define CREAT            8
#define LINK             9
#define UNLINK          10
#define WAITPID         11
#define CHDIR           12
#define TIME            13
#define MKNOD           14
#define CHMOD           15
#define CHOWN           16
#define BRK             17
#define STAT            18
#define LSEEK           19
#define GETPID          20
#define MOUNT           21
#define UMOUNT          22
#define SETUID          23
#define GETUID          24
#define STIME           25
#define PTRACE          26
#define ALARM           27
#define FSTAT           28
#define PAUSE           29
#define UTIME           30
#define ACCESS          33
#define SYNC            36
#define KILL            37
```

```
#define RENAME            38
#define MKDIR             39
#define RMDIR             40
#define DUP               41
#define PIPE              42
#define TIMES             43
#define SETGID            46
#define GETGID            47
#define SIGNAL            48
#define IOCTL             54
#define FCNTL             55
#define EXEC              59
#define UMASK             60
#define CHROOT            61
#define SETSID            62
#define GETPGRP           63

/* The following are not system calls, but are processed like them. */
#define KSIG              64    /* kernel detected a signal */
#define UNPAUSE           65    /* to MM or FS: check for EINTR */
#define REVIVE            67    /* to FS: revive a sleeping process */
#define TASK_REPLY        68    /* to FS: reply code from tty task */

/* Posix signal handling. */
#define SIGACTION         71
#define SIGSUSPEND        72
#define SIGPENDING        73
#define SIGPROCMASK       74
#define SIGRETURN         75

#define REBOOT            76
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/minix/com.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* System calls. */
#define SEND               1    /* function code for sending messages */
#define RECEIVE            2    /* function code for receiving messages */
#define BOTH               3    /* function code for SEND + RECEIVE */
#define ANY   (NR_PROCS+100)    /* receive(ANY, buf) accepts from any source */

/* Task numbers, function codes and reply codes. */

/* The values of several task numbers depend on whether they or other tasks
 * are enabled.  They are defined as (PREVIOUS_TASK - ENABLE_TASK) in general.
 * ENABLE_TASK is either 0 or 1, so a task either gets a new number, or gets
 * the same number as the previous task and is further unused.
 * The TTY task must always have the most negative number so that it is
 * initialized first.  Many of the TTY function codes are shared with other
 * tasks.
 */

#define TTY             (DL_ETH - 1)
                                /* terminal I/O class */
#       define CANCEL       0   /* general req to force a task to cancel */
```

```
#       define HARD_INT     2   /* fcn code for all hardware interrupts */
#       define DEV_READ     3   /* fcn code for reading from tty */
#       define DEV_WRITE    4   /* fcn code for writing to tty */
#       define DEV_IOCTL    5   /* fcn code for ioctl */
#       define DEV_OPEN     6   /* fcn code for opening tty */
#       define DEV_CLOSE    7   /* fcn code for closing tty */
#       define SCATTERED_IO 8   /* fcn code for multiple reads/writes */
#       define TTY_SETPGRP  9   /* fcn code for setpgroup */
#       define TTY_EXIT    10   /* a process group leader has exited */
#       define OPTIONAL_IO 16   /* modifier to DEV_* codes within vector */
#       define SUSPEND   -998   /* used in interrupts when tty has no data */

#define DL_ETH          (CDROM - ENABLE_NETWORKING)
                                /* networking task */

/* Message type for data link layer reqests. */
#       define DL_WRITE         3
#       define DL_WRITEV        4
#       define DL_READ          5
#       define DL_READV         6
#       define DL_INIT          7
#       define DL_STOP          8
#       define DL_GETSTAT       9

/* Message type for data link layer replies. */
#       define DL_INIT_REPLY    20
#       define DL_TASK_REPLY    21

#       define DL_PORT          m2_i1
#       define DL_PROC          m2_i2
#       define DL_COUNT         m2_i3
#       define DL_MODE          m2_l1
#       define DL_CLCK          m2_l2
#       define DL_ADDR          m2_p1
#       define DL_STAT          m2_l1

/* Bits in 'DL_STAT' field of DL replies. */
#       define DL_PACK_SEND     0x01
#       define DL_PACK_RECV     0x02
#       define DL_READ_IP       0x04

/* Bits in 'DL_MODE' field of DL requests. */
#       define DL_NOMODE        0x0
#       define DL_PROMISC_REQ   0x2
#       define DL_MULTI_REQ     0x4
#       define DL_BROAD_REQ     0x8

#       define NW_OPEN          DEV_OPEN
#       define NW_CLOSE         DEV_CLOSE
#       define NW_READ          DEV_READ
#       define NW_WRITE         DEV_WRITE
#       define NW_IOCTL         DEV_IOCTL
#       define NW_CANCEL        CANCEL

#define CDROM           (AUDIO - ENABLE_CDROM)
                                /* cd-rom device task */

#define AUDIO           (MIXER - ENABLE_AUDIO)
#define MIXER           (SCSI - ENABLE_AUDIO)
                                /* audio & mixer device tasks */
```

```

#define SCSI            (WINCHESTER - ENABLE_SCSI)
                                /* scsi device task */

#define WINCHESTER      (SYN_ALRM_TASK - ENABLE_WINI)
                                /* winchester (hard) disk class */

#define SYN_ALRM_TASK     -8    /* task to send CLOCK_INT messages */

#define IDLE              -7    /* task to run when there's nothing to run */

#define PRINTER           -6    /* printer I/O class */

#define FLOPPY            -5    /* floppy disk class */

#define MEM               -4    /* /dev/ram, /dev/(k)mem and /dev/null class */
#       define NULL_MAJOR  1    /* major device for /dev/null */
#       define RAM_DEV     0    /* minor device for /dev/ram */
#       define MEM_DEV     1    /* minor device for /dev/mem */
#       define KMEM_DEV    2    /* minor device for /dev/kmem */
#       define NULL_DEV    3    /* minor device for /dev/null */

#define CLOCK             -3    /* clock class */
#       define SET_ALARM   1    /* fcn code to CLOCK, set up alarm */
#       define GET_TIME    3    /* fcn code to CLOCK, get real time */
#       define SET_TIME    4    /* fcn code to CLOCK, set real time */
#       define GET_UPTIME  5    /* fcn code to CLOCK, get uptime */
#       define SET_SYNC_AL 6    /* fcn code to CLOCK, set up alarm which */
                                /* times out with a send */
#       define REAL_TIME   1    /* reply from CLOCK: here is real time */
#       define CLOCK_INT   HARD_INT
                                /* this code will only be sent by */
                                /* SYN_ALRM_TASK to a task that requested a */
                                /* synchronous alarm */

#define SYSTASK           -2    /* internal functions */
#       define SYS_XIT        1 /* fcn code for sys_xit(parent, proc) */
#       define SYS_GETSP      2 /* fcn code for sys_sp(proc, &new_sp) */
#       define SYS_OLDSIG     3 /* fcn code for sys_oldsig(proc, sig) */
#       define SYS_FORK       4 /* fcn code for sys_fork(parent, child) */
#       define SYS_NEWMAP     5 /* fcn code for sys_newmap(procno, map_ptr) */
#       define SYS_COPY       6 /* fcn code for sys_copy(ptr) */
#       define SYS_EXEC       7 /* fcn code for sys_exec(procno, new_sp) */
#       define SYS_TIMES      8 /* fcn code for sys_times(procno, bufptr) */
#       define SYS_ABORT      9 /* fcn code for sys_abort() */
#       define SYS_FRESH     10 /* fcn code for sys_fresh()  (Atari only) */
#       define SYS_KILL      11 /* fcn code for sys_kill(proc, sig) */
#       define SYS_GBOOT     12 /* fcn code for sys_gboot(procno, bootptr) */
#       define SYS_UMAP      13 /* fcn code for sys_umap(procno, etc) */
#       define SYS_MEM       14 /* fcn code for sys_mem() */
#       define SYS_TRACE     15 /* fcn code for sys_trace(req,pid,addr,data) */
#       define SYS_VCOPY     16 /* fnc code for sys_vcopy(src_proc, dest_proc,
                                   vcopy_s, vcopy_ptr) */
#       define SYS_SENDSIG   17 /* fcn code for sys_sendsig(&sigmsg) */
#       define SYS_SIGRETURN 18 /* fcn code for sys_sigreturn(&sigmsg) */
#       define SYS_ENDSIG    19 /* fcn code for sys_endsig(procno) */
#       define SYS_GETMAP    20 /* fcn code for sys_getmap(procno, map_ptr) */

#define HARDWARE          -1    /* used as source on interrupt generated msgs*/

```

```
/* Names of message fields for messages to CLOCK task. */
#define DELTA_TICKS    m6_l1    /* alarm interval in clock ticks */
#define FUNC_TO_CALL   m6_f1    /* pointer to function to call */
#define NEW_TIME       m6_l1    /* value to set clock to (SET_TIME) */
#define CLOCK_PROC_NR  m6_i1    /* which proc (or task) wants the alarm? */
#define SECONDS_LEFT   m6_l1    /* how many seconds were remaining */

/* Names of message fields used for messages to block and character tasks. */
#define DEVICE         m2_i1    /* major-minor device */
#define PROC_NR        m2_i2    /* which (proc) wants I/O? */
#define COUNT          m2_i3    /* how many bytes to transfer */
#define REQUEST        m2_i3    /* ioctl request code */
#define POSITION       m2_l1    /* file offset */
#define ADDRESS        m2_p1    /* core buffer address */

/* Names of message fields for messages to TTY task. */
#define TTY_LINE       DEVICE   /* message parameter: terminal line */
#define TTY_REQUEST    COUNT    /* message parameter: ioctl request code */
#define TTY_SPEK       POSITION /* message parameter: ioctl speed, erasing */
#define TTY_FLAGS      m2_l2    /* message parameter: ioctl tty mode */
#define TTY_PGRP       m2_i3    /* message parameter: process group */

/* Names of the message fields for QIC 02 status reply from tape driver */
#define TAPE_STAT0      m2_l1
#define TAPE_STAT1      m2_l2

/* Names of messages fields used in reply messages from tasks. */
#define REP_PROC_NR    m2_i1    /* # of proc on whose behalf I/O was done */
#define REP_STATUS     m2_i2    /* bytes transferred or error number */

/* Names of fields for copy message to SYSTASK. */
#define SRC_SPACE      m5_c1    /* T or D space (stack is also D) */
#define SRC_PROC_NR    m5_i1    /* process to copy from */
#define SRC_BUFFER     m5_l1    /* virtual address where data come from */
#define DST_SPACE      m5_c2    /* T or D space (stack is also D) */
#define DST_PROC_NR    m5_i2    /* process to copy to */
#define DST_BUFFER     m5_l2    /* virtual address where data go to */
#define COPY_BYTES     m5_l3    /* number of bytes to copy */

/* Field names for accounting, SYSTASK and miscellaneous. */
#define USER_TIME      m4_l1    /* user time consumed by process */
#define SYSTEM_TIME    m4_l2    /* system time consumed by process */
#define CHILD_UTIME    m4_l3    /* user time consumed by process' children */
#define CHILD_STIME    m4_l4    /* sys time consumed by process' children */
#define BOOT_TICKS     m4_l5    /* number of clock ticks since boot time */

#define PROC1          m1_i1    /* indicates a process */
#define PROC2          m1_i2    /* indicates a process */
#define PID            m1_i3    /* process id passed from MM to kernel */
#define STACK_PTR      m1_p1    /* used for stack ptr in sys_exec, sys_getsp */
#define PR             m6_i1    /* process number for sys_sig */
#define SIGNUM         m6_i2    /* signal number for sys_sig */
#define FUNC           m6_f1    /* function pointer for sys_sig */
#define MEM_PTR        m1_p1    /* tells where memory map is for sys_newmap */
#define NAME_PTR       m1_p2    /* tells where program name is for dmp */
#define IP_PTR         m1_p3    /* initial value for ip after exec */
#define SIG_PROC       m2_i1    /* process number for inform */
#define SIG_MAP        m2_l1    /* used by kernel for passing signal bit map */
#define SIG_MSG_PTR    m1_i1    /* pointer to info to build sig catch stack */
#define SIG_CTXT_PTR   m1_p1    /* pointer to info to restore signal context */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  include/minix/boot.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* boot.h */

#ifndef _BOOT_H
#define _BOOT_H

/* Redefine root and root image devices as variables.
 * This keeps the diffs small but may cause future confusion.
 */
#define ROOT_DEV   (boot_parameters.bp_rootdev)
#define IMAGE_DEV  (boot_parameters.bp_ramimagedev)

/* Device numbers of RAM, floppy and hard disk devices.
 * h/com.h defines RAM_DEV but only as the minor number.
 */
#define DEV_FD0   0x200
#define DEV_HD0   0x300
#define DEV_RAM   0x100
#define DEV_SCSI  0x700 /* Atari TT only */

/* Structure to hold boot parameters. */
struct bparam_s
{
  dev_t bp_rootdev;
  dev_t bp_ramimagedev;
  unsigned short bp_ramsize;
  unsigned short bp_processor;
};

extern struct bparam_s boot_parameters;
#endif /* _BOOT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  include/minix/keymap.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/*      keymap.h - defines for keymapping              Author: Marcus Hampel
 */
#ifndef _SYS__KEYMAP_H
#define _SYS__KEYMAP_H

#define C(c)    ((c) & 0x1F)    /* Map to control code          */
#define A(c)    ((c) | 0x80)    /* Set eight bit (ALT)          */
#define CA(c)   A(C(c))         /* Control-Alt                  */
#define L(c)    ((c) | HASCAPS) /* Add "Caps Lock has effect" attribute */

#define EXT     0x0100          /* Normal function keys         */
#define CTRL    0x0200          /* Control key                  */
#define SHIFT   0x0400          /* Shift key                    */
#define ALT     0x0800          /* Alternate key                */
#define EXTKEY  0x1000          /* extended keycode             */
```

```
#define HASCAPS 0x8000          /* Caps Lock has effect         */

/* Numeric keypad */
#define HOME    (0x01 + EXT)
#define END     (0x02 + EXT)
#define UP      (0x03 + EXT)
#define DOWN    (0x04 + EXT)
#define LEFT    (0x05 + EXT)
#define RIGHT   (0x06 + EXT)
#define PGUP    (0x07 + EXT)
#define PGDN    (0x08 + EXT)
#define MID     (0x09 + EXT)
#define NMIN    (0x0A + EXT)
#define PLUS    (0x0B + EXT)
#define INSRT   (0x0C + EXT)

/* Alt + Numeric keypad */
#define AHOME   (0x01 + ALT)
#define AEND    (0x02 + ALT)
#define AUP     (0x03 + ALT)
#define ADOWN   (0x04 + ALT)
#define ALEFT   (0x05 + ALT)
#define ARIGHT  (0x06 + ALT)
#define APGUP   (0x07 + ALT)
#define APGDN   (0x08 + ALT)
#define AMID    (0x09 + ALT)
#define ANMIN   (0x0A + ALT)
#define APLUS   (0x0B + ALT)
#define AINSRT  (0x0C + ALT)

/* Ctrl + Numeric keypad */
#define CHOME   (0x01 + CTRL)
#define CEND    (0x02 + CTRL)
#define CUP     (0x03 + CTRL)
#define CDOWN   (0x04 + CTRL)
#define CLEFT   (0x05 + CTRL)
#define CRIGHT  (0x06 + CTRL)
#define CPGUP   (0x07 + CTRL)
#define CPGDN   (0x08 + CTRL)
#define CMID    (0x09 + CTRL)
#define CNMIN   (0x0A + CTRL)
#define CPLUS   (0x0B + CTRL)
#define CINSRT  (0x0C + CTRL)

/* Lock keys */
#define CALOCK  (0x0D + EXT)    /* caps lock    */
#define NLOCK   (0x0E + EXT)    /* number lock  */
#define SLOCK   (0x0F + EXT)    /* scroll lock  */

/* Function keys */
#define F1      (0x10 + EXT)
#define F2      (0x11 + EXT)
#define F3      (0x12 + EXT)
#define F4      (0x13 + EXT)
#define F5      (0x14 + EXT)
#define F6      (0x15 + EXT)
#define F7      (0x16 + EXT)
#define F8      (0x17 + EXT)
#define F9      (0x18 + EXT)
#define F10     (0x19 + EXT)
```

```
#define F11     (0x1A + EXT)
#define F12     (0x1B + EXT)

/* Alt+Fn */
#define AF1     (0x10 + ALT)
#define AF2     (0x11 + ALT)
#define AF3     (0x12 + ALT)
#define AF4     (0x13 + ALT)
#define AF5     (0x14 + ALT)
#define AF6     (0x15 + ALT)
#define AF7     (0x16 + ALT)
#define AF8     (0x17 + ALT)
#define AF9     (0x18 + ALT)
#define AF10    (0x19 + ALT)
#define AF11    (0x1A + ALT)
#define AF12    (0x1B + ALT)

/* Ctrl+Fn */
#define CF1     (0x10 + CTRL)
#define CF2     (0x11 + CTRL)
#define CF3     (0x12 + CTRL)
#define CF4     (0x13 + CTRL)
#define CF5     (0x14 + CTRL)
#define CF6     (0x15 + CTRL)
#define CF7     (0x16 + CTRL)
#define CF8     (0x17 + CTRL)
#define CF9     (0x18 + CTRL)
#define CF10    (0x19 + CTRL)
#define CF11    (0x1A + CTRL)
#define CF12    (0x1B + CTRL)

/* Shift+Fn */
#define SF1     (0x10 + SHIFT)
#define SF2     (0x11 + SHIFT)
#define SF3     (0x12 + SHIFT)
#define SF4     (0x13 + SHIFT)
#define SF5     (0x14 + SHIFT)
#define SF6     (0x15 + SHIFT)
#define SF7     (0x16 + SHIFT)
#define SF8     (0x17 + SHIFT)
#define SF9     (0x18 + SHIFT)
#define SF10    (0x19 + SHIFT)
#define SF11    (0x1A + SHIFT)
#define SF12    (0x1B + SHIFT)

/* Alt+Shift+Fn */
#define ASF1    (0x10 + ALT + SHIFT)
#define ASF2    (0x11 + ALT + SHIFT)
#define ASF3    (0x12 + ALT + SHIFT)
#define ASF4    (0x13 + ALT + SHIFT)
#define ASF5    (0x14 + ALT + SHIFT)
#define ASF6    (0x15 + ALT + SHIFT)
#define ASF7    (0x16 + ALT + SHIFT)
#define ASF8    (0x17 + ALT + SHIFT)
#define ASF9    (0x18 + ALT + SHIFT)
#define ASF10   (0x19 + ALT + SHIFT)
#define ASF11   (0x1A + ALT + SHIFT)
#define ASF12   (0x1B + ALT + SHIFT)

#define MAP_COLS        6       /* Number of columns in keymap */
```

```
#define NR_SCAN_CODES   0x80    /* Number of scan codes (rows in keymap) */

typedef unsigned short keymap_t[NR_SCAN_CODES * MAP_COLS];

#define KEY_MAGIC       "KMAZ"  /* Magic number of keymap file */

#endif /* _SYS__KEYMAP_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/minix/partition.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*      minix/partition.h                              Author: Kees J. Bot
 *                                                             7 Dec 1995
 * Place of a partition on disk and the disk geometry,
 * for use with the DIOCGETP and DIOCSETP ioctl's.
 */
#ifndef _MINIX__PARTITION_H
#define _MINIX__PARTITION_H

struct partition {
  u32_t base;           /* byte offset to the partition start */
  u32_t size;           /* number of bytes in the partition */
  unsigned cylinders;   /* disk geometry */
  unsigned heads;
  unsigned sectors;
};
#endif /* _MINIX__PARTITION_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 include/ibm/partition.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Description of entry in partition table.  */
#ifndef _PARTITION_H
#define _PARTITION_H

struct part_entry {
  unsigned char bootind;        /* boot indicator 0/ACTIVE_FLAG  */
  unsigned char start_head;     /* head value for first sector   */
  unsigned char start_sec;      /* sector value + cyl bits for first sector */
  unsigned char start_cyl;      /* track value for first sector  */
  unsigned char sysind;         /* system indicator              */
  unsigned char last_head;      /* head value for last sector    */
  unsigned char last_sec;       /* sector value + cyl bits for last sector */
  unsigned char last_cyl;       /* track value for last sector   */
  unsigned long lowsec;         /* logical first sector          */
  unsigned long size;           /* size of partition in sectors  */
};

#define ACTIVE_FLAG     0x80    /* value for active in bootind field (hd0) */
#define NR_PARTITIONS   4       /* number of entries in partition table */
#define PART_TABLE_OFF  0x1BE   /* offset of partition table in boot sector */
```

```

/* Partition types. */
#define MINIX_PART      0x81    /* Minix partition type */
#define NO_PART         0x00    /* unused entry */
#define OLD_MINIX_PART  0x80    /* created before 1.4b, obsolete */
#define EXT_PART        0x05    /* extended partition */

#endif /* _PARTITION_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/kernel.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This is the master header for the kernel.  It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM            1    /* tell headers that this is the kernel */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>       /* MUST be first */
#include <ansi.h>               /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/syslib.h>

#include <string.h>
#include <limits.h>
#include <errno.h>

#include "const.h"
#include "type.h"
#include "proto.h"
#include "glo.h"
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* General constants used by the kernel. */

#if (CHIP == INTEL)

#define K_STACK_BYTES   1024    /* how many bytes for the kernel stack */

#define INIT_PSW      0x0200    /* initial psw */
#define INIT_TASK_PSW 0x1200    /* initial psw for tasks (with IOPL 1) */
#define TRACEBIT       0x100    /* OR this with psw in proc[] for tracing */
#define SETPSW(rp, new) /* permits only certain bits to be set */ \
        ((rp)->p_reg.psw = (rp)->p_reg.psw & ~0xCD5 | (new) & 0xCD5)

/* Initial sp for mm, fs and init.
 *      2 bytes for short jump
 *      2 bytes unused
```

```
 *      3 words for init_org[] used by fs only
 *      3 words for real mode debugger trap (actually needs 1 more)
 *      3 words for save and restart temporaries
 *      3 words for interrupt
 * Leave no margin, to flush bugs early.
 */
#define INIT_SP (2 + 2 + 3 * 2 + 3 * 2 + 3 * 2 + 3 * 2)

#define HCLICK_SHIFT       4    /* log2 of HCLICK_SIZE */
#define HCLICK_SIZE       16    /* hardware segment conversion magic */
#if CLICK_SIZE >= HCLICK_SIZE
#define click_to_hclick(n) ((n) << (CLICK_SHIFT - HCLICK_SHIFT))
#else
#define click_to_hclick(n) ((n) >> (HCLICK_SHIFT - CLICK_SHIFT))
#endif
#define hclick_to_physb(n) ((phys_bytes) (n) << HCLICK_SHIFT)
#define physb_to_hclick(n) ((n) >> HCLICK_SHIFT)

/* Interrupt vectors defined/reserved by processor. */
#define DIVIDE_VECTOR      0    /* divide error */
#define DEBUG_VECTOR       1    /* single step (trace) */
#define NMI_VECTOR         2    /* non-maskable interrupt */
#define BREAKPOINT_VECTOR  3    /* software breakpoint */
#define OVERFLOW_VECTOR    4    /* from INTO */

/* Fixed system call vector. */
#define SYS_VECTOR        32    /* system calls are made with int SYSVEC */
#define SYS386_VECTOR     33    /* except 386 system calls use this */
#define LEVEL0_VECTOR     34    /* for execution of a function at level 0 */

/* Suitable irq bases for hardware interrupts.  Reprogram the 8259(s) from
 * the PC BIOS defaults since the BIOS doesn't respect all the processor's
 * reserved vectors (0 to 31).
 */
#define BIOS_IRQ0_VEC   0x08    /* base of IRQ0-7 vectors used by BIOS */
#define BIOS_IRQ8_VEC   0x70    /* base of IRQ8-15 vectors used by BIOS */
#define IRQ0_VECTOR     0x28    /* more or less arbitrary, but > SYS_VECTOR */
#define IRQ8_VECTOR     0x30    /* together for simplicity */

/* Hardware interrupt numbers. */
#define NR_IRQ_VECTORS    16
#define CLOCK_IRQ          0
#define KEYBOARD_IRQ       1
#define CASCADE_IRQ        2    /* cascade enable for 2nd AT controller */
#define ETHER_IRQ          3    /* default ethernet interrupt vector */
#define SECONDARY_IRQ      3    /* RS232 interrupt vector for port 2 */
#define RS232_IRQ          4    /* RS232 interrupt vector for port 1 */
#define XT_WINI_IRQ        5    /* xt winchester */
#define FLOPPY_IRQ         6    /* floppy disk */
#define PRINTER_IRQ        7
#define AT_WINI_IRQ       14    /* at winchester */

/* Interrupt number to hardware vector. */
#define BIOS_VECTOR(irq)        \
        (((irq) < 8 ? BIOS_IRQ0_VEC : BIOS_IRQ8_VEC) + ((irq) & 0x07))
#define VECTOR(irq)     \
        (((irq) < 8 ? IRQ0_VECTOR : IRQ8_VECTOR) + ((irq) & 0x07))

/* BIOS hard disk parameter vectors. */
#define WINI_0_PARM_VEC 0x41
```

```
#define WINI_1_PARM_VEC 0x46

/* 8259A interrupt controller ports. */
#define INT_CTL         0x20    /* I/O port for interrupt controller */
#define INT_CTLMASK     0x21    /* setting bits in this port disables ints */
#define INT2_CTL        0xA0    /* I/O port for second interrupt controller */
#define INT2_CTLMASK    0xA1    /* setting bits in this port disables ints */

/* Magic numbers for interrupt controller. */
#define ENABLE          0x20    /* code used to re-enable after an interrupt */

/* Sizes of memory tables. */
#define NR_MEMS            3    /* number of chunks of memory */

/* Miscellaneous ports. */
#define PCR             0x65    /* Planar Control Register */
#define PORT_B          0x61    /* I/O port for 8255 port B (kbd, beeper...) */
#define TIMER0          0x40    /* I/O port for timer channel 0 */
#define TIMER2          0x42    /* I/O port for timer channel 2 */
#define TIMER_MODE      0x43    /* I/O port for timer mode control */

#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)

#define K_STACK_BYTES   1024    /* how many bytes for the kernel stack */

/* Sizes of memory tables. */
#define NR_MEMS            2    /* number of chunks of memory */

/* p_reg contains: d0-d7, a0-a6,   in that order. */
#define NR_REGS           15    /* number of general regs in each proc slot */

#define TRACEBIT      0x8000    /* or this with psw in proc[] for tracing */
#define SETPSW(rp, new)         /* permits only certain bits to be set */ \
        ((rp)->p_reg.psw = (rp)->p_reg.psw & ~0xFF | (new) & 0xFF)

#define MEM_BYTES  0xffffffff   /* memory size for /dev/mem */

#ifdef __ACK__
#define FSTRUCOPY
#endif

#endif /* (CHIP == M68000) */

/* The following items pertain to the scheduling queues. */
#define TASK_Q             0    /* ready tasks are scheduled via queue 0 */
#define SERVER_Q           1    /* ready servers are scheduled via queue 1 */
#define USER_Q             2    /* ready users are scheduled via queue 2 */

#if (MACHINE == ATARI)
#define SHADOW_Q           3    /* runnable, but shadowed processes */
#define NQ                 4    /* # of scheduling queues */
#else
#define NQ                 3    /* # of scheduling queues */
#endif

/* Env_parse() return values. */
#define EP_UNSET        0       /* variable not set */
#define EP_OFF          1       /* var = off */
```

```
#define EP_ON           2       /* var = on (or field left blank) */
#define EP_SET          3       /* var = 1:2:3 (nonblank field) */

/* To translate an address in kernel space to a physical address.  This is
 * the same as umap(proc_ptr, D, vir, sizeof(*vir)), but a lot less costly.
 */
#define vir2phys(vir)   (data_base + (vir_bytes) (vir))

#define printf        printk    /* the kernel really uses printk, not printf */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#ifndef TYPE_H
#define TYPE_H

typedef _PROTOTYPE( void task_t, (void) );
typedef _PROTOTYPE( int (*rdwt_t), (message *m_ptr) );
typedef _PROTOTYPE( void (*watchdog_t), (void) );

struct tasktab {
  task_t *initial_pc;
  int stksize;
  char name[8];
};

struct memory {
  phys_clicks base;
  phys_clicks size;
};

/* Administration for clock polling. */
struct milli_state {
  unsigned long accum_count;    /* accumulated clock ticks */
  unsigned prev_count;          /* previous clock value */
};

#if (CHIP == INTEL)
typedef unsigned port_t;
typedef unsigned segm_t;
typedef unsigned reg_t;         /* machine register */

/* The stack frame layout is determined by the software, but for efficiency
 * it is laid out so the assembly code to use it is as simple as possible.
 * 80286 protected mode and all real modes use the same frame, built with
 * 16-bit registers.  Real mode lacks an automatic stack switch, so little
 * is lost by using the 286 frame for it.  The 386 frame differs only in
 * having 32-bit registers and more segment registers.  The same names are
 * used for the larger registers to avoid differences in the code.
 */
struct stackframe_s {           /* proc_ptr points here */
#if _WORD_SIZE == 4
  u16_t gs;                     /* last item pushed by save */
  u16_t fs;                     /*  ^ */
#endif
  u16_t es;                     /*  | */
  u16_t ds;                     /*  | */
  reg_t di;                     /* di through cx are not accessed in C */
```

```
  reg_t si;                    /* order is to match pusha/popa */
  reg_t fp;                    /* bp */
  reg_t st;                    /* hole for another copy of sp */
  reg_t bx;                    /*  | */
  reg_t dx;                    /*  | */
  reg_t cx;                    /*  | */
  reg_t retreg;                /* ax and above are all pushed by save */
  reg_t retadr;                /* return address for assembly code save() */
  reg_t pc;                    /*  ^  last item pushed by interrupt */
  reg_t cs;                    /*  | */
  reg_t psw;                   /*  | */
  reg_t sp;                    /*  | */
  reg_t ss;                    /* these are pushed by CPU during interrupt */
};

struct segdesc_s {             /* segment descriptor for protected mode */
  u16_t limit_low;
  u16_t base_low;
  u8_t base_middle;
  u8_t access;                 /* |P|DL|1|X|E|R|A| */
#if _WORD_SIZE == 4
  u8_t granularity;            /* |G|X|0|A|LIMT| */
  u8_t base_high;
#else
  u16_t reserved;
#endif
};

typedef _PROTOTYPE( int (*irq_handler_t), (int irq) );

#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
typedef _PROTOTYPE( void (*dmaint_t), (void) );

typedef u32_t reg_t;           /* machine register */

/* The name and fields of this struct were chosen for PC compatibility. */
struct stackframe_s {
  reg_t retreg;                /* d0 */
  reg_t d1;
  reg_t d2;
  reg_t d3;
  reg_t d4;
  reg_t d5;
  reg_t d6;
  reg_t d7;
  reg_t a0;
  reg_t a1;
  reg_t a2;
  reg_t a3;
  reg_t a4;
  reg_t a5;
  reg_t fp;                    /* also known as a6 */
  reg_t sp;                    /* also known as a7 */
  reg_t pc;
  u16_t psw;
  u16_t dummy;                 /* make size multiple of reg_t for system.c */
};
```

```
struct fsave {
  struct cpu_state {
        u16_t i_format;
        u32_t i_addr;
        u16_t i_state[4];
  } cpu_state;
  struct state_frame {
        u8_t frame_type;
        u8_t frame_size;
        u16_t reserved;
        u8_t frame[212];
  } state_frame;
  struct fpp_model {
        u32_t fpcr;
        u32_t fpsr;
        u32_t fpiar;
        struct fpN {
                u32_t high;
                u32_t low;
                u32_t mid;
        } fpN[8];
  } fpp_model;
};
#endif /* (CHIP == M68000) */

#endif /* TYPE_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Function prototypes. */

#ifndef PROTO_H
#define PROTO_H

/* Struct declarations. */
struct proc;
struct tty;

/* at_wini.c, wini.c */
_PROTOTYPE( void winchester_task, (void)                                  );
_PROTOTYPE( void at_winchester_task, (void)                               );

/* clock.c */
_PROTOTYPE( void clock_task, (void)                                       );
_PROTOTYPE( void clock_stop, (void)                                       );
_PROTOTYPE( clock_t get_uptime, (void)                                    );
_PROTOTYPE( void syn_alrm_task, (void)                                    );

/* dmp.c */
_PROTOTYPE( void map_dmp, (void)                                          );
_PROTOTYPE( void p_dmp, (void)                                            );
_PROTOTYPE( void reg_dmp, (struct proc *rp)                               );

/* dp8390.c */
```

```
_PROTOTYPE( void dp8390_task, (void)                                    );
_PROTOTYPE( void dp_dump, (void)                                        );
_PROTOTYPE( void dp8390_stop, (void)                                    );

/* floppy.c, stfloppy.c */
_PROTOTYPE( void floppy_task, (void)                                    );
_PROTOTYPE( void floppy_stop, (void)                                    );

/* main.c, stmain.c */
_PROTOTYPE( void main, (void)                                           );
_PROTOTYPE( void panic, (const char *s, int n)                          );

/* memory.c */
_PROTOTYPE( void mem_task, (void)                                       );

/* misc.c */
_PROTOTYPE( int env_parse, (char *env, char *fmt, int field,
                        long *param, long min, long max)                );

/* printer.c, stprint.c */
_PROTOTYPE( void printer_task, (void)                                   );

/* proc.c */
_PROTOTYPE( void interrupt, (int task)                                  );
_PROTOTYPE( int lock_mini_send, (struct proc *caller_ptr,
                int dest, message *m_ptr)                               );
_PROTOTYPE( void lock_pick_proc, (void)                                 );
_PROTOTYPE( void lock_ready, (struct proc *rp)                          );
_PROTOTYPE( void lock_sched, (void)                                     );
_PROTOTYPE( void lock_unready, (struct proc *rp)                        );
_PROTOTYPE( int sys_call, (int function, int src_dest, message *m_ptr)  );
_PROTOTYPE( void unhold, (void)                                         );

/* rs232.c */
_PROTOTYPE( void rs_init, (struct tty *tp)                              );

/* system.c */
_PROTOTYPE( void cause_sig, (int proc_nr, int sig_nr)                   );
_PROTOTYPE( void inform, (void)                                         );
_PROTOTYPE( phys_bytes numap, (int proc_nr, vir_bytes vir_addr,
                vir_bytes bytes)                                        );
_PROTOTYPE( void sys_task, (void)                                       );
_PROTOTYPE( phys_bytes umap, (struct proc *rp, int seg, vir_bytes vir_addr,
                vir_bytes bytes)                                        );

/* tty.c */
_PROTOTYPE( void handle_events, (struct tty *tp)                        );
_PROTOTYPE( void sigchar, (struct tty *tp, int sig)                     );
_PROTOTYPE( void tty_task, (void)                                       );
_PROTOTYPE( int in_process, (struct tty *tp, char *buf, int count)      );
_PROTOTYPE( void out_process, (struct tty *tp, char *bstart, char *bpos,
                                char *bend, int *icount, int *ocount)   );
_PROTOTYPE( void tty_wakeup, (clock_t now)                              );
_PROTOTYPE( void tty_reply, (int code, int replyee, int proc_nr,
                                                        int status)     );
_PROTOTYPE( void tty_devnop, (struct tty *tp)                           );

/* library */
_PROTOTYPE( void *memcpy, (void *_s1, const void *_s2, size_t _n)       );
```

```
#if (CHIP == INTEL)

/* clock.c */
_PROTOTYPE( void milli_start, (struct milli_state *msp)                 );
_PROTOTYPE( unsigned milli_elapsed, (struct milli_state *msp)           );
_PROTOTYPE( void milli_delay, (unsigned millisec)                       );

/* console.c */
_PROTOTYPE( void cons_stop, (void)                                      );
_PROTOTYPE( void putk, (int c)                                          );
_PROTOTYPE( void scr_init, (struct tty *tp)                             );
_PROTOTYPE( void toggle_scroll, (void)                                  );
_PROTOTYPE( int con_loadfont, (phys_bytes user_phys)                    );
_PROTOTYPE( void select_console, (int cons_line)                        );

/* cstart.c */
_PROTOTYPE( void cstart, (U16_t cs, U16_t ds, U16_t mcs, U16_t mds,
                                U16_t parmoff, U16_t parmsize)          );
_PROTOTYPE( char *k_getenv, (char *name)                                );

/* exception.c */
_PROTOTYPE( void exception, (unsigned vec_nr)                           );

/* i8259.c */
_PROTOTYPE( irq_handler_t get_irq_handler, (int irq)                    );
_PROTOTYPE( void put_irq_handler, (int irq, irq_handler_t handler)      );
_PROTOTYPE( void intr_init, (int mine)                                  );

/* keyboard.c */
_PROTOTYPE( void kb_init, (struct tty *tp)                              );
_PROTOTYPE( int kbd_loadmap, (phys_bytes user_phys)                     );
_PROTOTYPE( void wreboot, (int how)                                     );

/* klib*.s */
_PROTOTYPE( void bios13, (void)                                         );
_PROTOTYPE( phys_bytes check_mem, (phys_bytes base, phys_bytes size)    );
_PROTOTYPE( void cp_mess, (int src,phys_clicks src_clicks,vir_bytes src_offset,
                phys_clicks dst_clicks, vir_bytes dst_offset)           );
_PROTOTYPE( int in_byte, (port_t port)                                  );
_PROTOTYPE( int in_word, (port_t port)                                  );
_PROTOTYPE( void lock, (void)                                           );
_PROTOTYPE( void unlock, (void)                                         );
_PROTOTYPE( void enable_irq, (unsigned irq)                             );
_PROTOTYPE( int disable_irq, (unsigned irq)                             );
_PROTOTYPE( u16_t mem_rdw, (segm_t segm, vir_bytes offset)              );
_PROTOTYPE( void out_byte, (port_t port, int value)                     );
_PROTOTYPE( void out_word, (port_t port, int value)                     );
_PROTOTYPE( void phys_copy, (phys_bytes source, phys_bytes dest,
                phys_bytes count)                                       );
_PROTOTYPE( void port_read, (unsigned port, phys_bytes destination,
                unsigned bytcount)                                      );
_PROTOTYPE( void port_read_byte, (unsigned port, phys_bytes destination,
                unsigned bytcount)                                      );
_PROTOTYPE( void port_write, (unsigned port, phys_bytes source,
                unsigned bytcount)                                      );
_PROTOTYPE( void port_write_byte, (unsigned port, phys_bytes source,
                unsigned bytcount)                                      );
_PROTOTYPE( void reset, (void)                                          );
_PROTOTYPE( void vid_vid_copy, (unsigned src, unsigned dst, unsigned count));
_PROTOTYPE( void mem_vid_copy, (u16_t *src, unsigned dst, unsigned count));
```

```
_PROTOTYPE( void level0, (void (*func)(void))                         );
_PROTOTYPE( void monitor, (void)                                      );

/* misc.c */
_PROTOTYPE( void mem_init, (void)                                     );

/* mpx*.s */
_PROTOTYPE( void idle_task, (void)                                    );
_PROTOTYPE( void restart, (void)                                      );

/* The following are never called from C (pure asm procs). */

/* Exception handlers (real or protected mode), in numerical order. */
void _PROTOTYPE( int00, (void) ), _PROTOTYPE( divide_error, (void) );
void _PROTOTYPE( int01, (void) ), _PROTOTYPE( single_step_exception, (void) );
void _PROTOTYPE( int02, (void) ), _PROTOTYPE( nmi, (void) );
void _PROTOTYPE( int03, (void) ), _PROTOTYPE( breakpoint_exception, (void) );
void _PROTOTYPE( int04, (void) ), _PROTOTYPE( overflow, (void) );
void _PROTOTYPE( int05, (void) ), _PROTOTYPE( bounds_check, (void) );
void _PROTOTYPE( int06, (void) ), _PROTOTYPE( inval_opcode, (void) );
void _PROTOTYPE( int07, (void) ), _PROTOTYPE( copr_not_available, (void) );
void                              _PROTOTYPE( double_fault, (void) );
void                              _PROTOTYPE( copr_seg_overrun, (void) );
void                              _PROTOTYPE( inval_tss, (void) );
void                              _PROTOTYPE( segment_not_present, (void) );
void                              _PROTOTYPE( stack_exception, (void) );
void                              _PROTOTYPE( general_protection, (void) );
void                              _PROTOTYPE( page_fault, (void) );
void                              _PROTOTYPE( copr_error, (void) );

/* Hardware interrupt handlers. */
_PROTOTYPE( void hwint00, (void) );
_PROTOTYPE( void hwint01, (void) );
_PROTOTYPE( void hwint02, (void) );
_PROTOTYPE( void hwint03, (void) );
_PROTOTYPE( void hwint04, (void) );
_PROTOTYPE( void hwint05, (void) );
_PROTOTYPE( void hwint06, (void) );
_PROTOTYPE( void hwint07, (void) );
_PROTOTYPE( void hwint08, (void) );
_PROTOTYPE( void hwint09, (void) );
_PROTOTYPE( void hwint10, (void) );
_PROTOTYPE( void hwint11, (void) );
_PROTOTYPE( void hwint12, (void) );
_PROTOTYPE( void hwint13, (void) );
_PROTOTYPE( void hwint14, (void) );
_PROTOTYPE( void hwint15, (void) );

/* Software interrupt handlers, in numerical order. */
_PROTOTYPE( void trp, (void) );
_PROTOTYPE( void s_call, (void) ), _PROTOTYPE( p_s_call, (void) );
_PROTOTYPE( void level0_call, (void) );

/* printer.c */
_PROTOTYPE( void pr_restart, (void)                                   );

/* protect.c */
_PROTOTYPE( void prot_init, (void)                                    );
_PROTOTYPE( void init_codeseg, (struct segdesc_s *segdp, phys_bytes base,
                phys_bytes size, int privilege)                       );
```

```
_PROTOTYPE( void init_dataseg, (struct segdesc_s *segdp, phys_bytes base,
                phys_bytes size, int privilege)                            );
_PROTOTYPE( phys_bytes seg2phys, (U16_t seg)                               );
_PROTOTYPE( void enable_iop, (struct proc *pp)                             );

/* pty.c */
_PROTOTYPE( void do_pty, (struct tty *tp, message *m_ptr)                  );
_PROTOTYPE( void pty_init, (struct tty *tp)                                );

/* system.c */
_PROTOTYPE( void alloc_segments, (struct proc *rp)                         );

#endif /* (CHIP == INTEL) */

#endif /* PROTO_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Global variables used in the kernel. */

/* EXTERN is defined as extern except in table.c. */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

/* Kernel memory. */
EXTERN phys_bytes code_base;    /* base of kernel code */
EXTERN phys_bytes data_base;    /* base of kernel data */

/* Low level interrupt communications. */
EXTERN struct proc *held_head;  /* head of queue of held-up interrupts */
EXTERN struct proc *held_tail;  /* tail of queue of held-up interrupts */
EXTERN unsigned char k_reenter; /* kernel reentry count (entry count less 1)*/

/* Process table.  Here to stop too many things having to include proc.h. */
EXTERN struct proc *proc_ptr;   /* pointer to currently running process */

/* Signals. */
EXTERN int sig_procs;           /* number of procs with p_pending != 0 */

/* Memory sizes. */
EXTERN struct memory mem[NR_MEMS];      /* base and size of chunks of memory */
EXTERN phys_clicks tot_mem_size;        /* total system memory size */

/* Miscellaneous. */
extern u16_t sizes[];           /* table filled in by boot monitor */
extern struct tasktab tasktab[];/* initialized in table.c, so extern here */
extern char *t_stack[];         /* initialized in table.c, so extern here */
EXTERN unsigned lost_ticks;     /* clock ticks counted outside the clock task */
EXTERN clock_t tty_timeout;     /* time to wake up the TTY task */
EXTERN int current;             /* currently visible console */

```

```
#if (CHIP == INTEL)

/* Machine type. */
EXTERN int pc_at;               /* PC-AT compatible hardware interface */
EXTERN int ps_mca;              /* PS/2 with Micro Channel */
EXTERN unsigned int processor;  /* 86, 186, 286, 386, ... */
#if _WORD_SIZE == 2
EXTERN int protected_mode;      /* nonzero if running in Intel protected mode*/
#else
#define protected_mode  1       /* 386 mode implies protected mode */
#endif

/* Video card types. */
EXTERN int ega;                 /* nonzero if console is EGA or VGA */
EXTERN int vga;                 /* nonzero if console is VGA */

/* Memory sizes. */
EXTERN unsigned ext_memsize;    /* initialized by assembler startup code */
EXTERN unsigned low_memsize;

/* Miscellaneous. */
EXTERN irq_handler_t irq_table[NR_IRQ_VECTORS];
EXTERN int irq_use;             /* bit map of all in-use irq's */
EXTERN reg_t mon_ss, mon_sp;    /* monitor stack */
EXTERN int mon_return;          /* true if return to the monitor possible */
EXTERN phys_bytes reboot_code;  /* program for the boot monitor */

/* Variables that are initialized elsewhere are just extern here. */
extern struct segdesc_s gdt[];  /* global descriptor table for protected mode*/

EXTERN _PROTOTYPE( void (*level0_func), (void) );
#endif /* (CHIP == INTEL) */

#if (CHIP == M68000)
/* Variables that are initialized elsewhere are just extern here. */
extern int keypad;              /* Flag for keypad mode */
extern int app_mode;            /* Flag for arrow key application mode */
extern int STdebKey;            /* nonzero if ctl-alt-Fx detected */
extern struct tty *cur_cons;    /* virtual cons currently displayed */
extern unsigned char font8[];   /* 8 pixel wide font table (initialized) */
extern unsigned char font12[];  /* 12 pixel wide font table (initialized) */
extern unsigned char font16[];  /* 16 pixel wide font table (initialized) */
extern unsigned short resolution; /* screen res; ST_RES_LOW..TT_RES_HIGH */
#endif
```

++

src/kernel/proc.h

++

```
#ifndef PROC_H
#define PROC_H

/* Here is the declaration of the process table.  It contains the process'
 * registers, memory map, accounting, and message send/receive information.
 * Many assembly code routines reference fields in it.  The offsets to these
 * fields are defined in the assembler include file sconst.h.  When changing
 * 'proc', be sure to change sconst.h to match.
 */

```

```
struct proc {
  struct stackframe_s p_reg;    /* process' registers saved in stack frame */

#if (CHIP == INTEL)
  reg_t p_ldt_sel;              /* selector in gdt giving ldt base and limit*/
  struct segdesc_s p_ldt[2];    /* local descriptors for code and data */
                                /* 2 is LDT_SIZE - avoid include protect.h */
#endif /* (CHIP == INTEL) */

  reg_t *p_stguard;             /* stack guard word */

  int p_nr;                     /* number of this process (for fast access) */

  int p_int_blocked;            /* nonzero if int msg blocked by busy task */
  int p_int_held;               /* nonzero if int msg held by busy syscall */
  struct proc *p_nextheld;      /* next in chain of held-up int processes */

  int p_flags;                  /* P_SLOT_FREE, SENDING, RECEIVING, etc. */
  struct mem_map p_map[NR_SEGS];/* memory map */
  pid_t p_pid;                  /* process id passed in from MM */

  clock_t user_time;            /* user time in ticks */
  clock_t sys_time;             /* sys time in ticks */
  clock_t child_utime;          /* cumulative user time of children */
  clock_t child_stime;          /* cumulative sys time of children */
  clock_t p_alarm;              /* time of next alarm in ticks, or 0 */

  struct proc *p_callerq;       /* head of list of procs wishing to send */
  struct proc *p_sendlink;      /* link to next proc wishing to send */
  message *p_messbuf;           /* pointer to message buffer */
  int p_getfrom;                /* from whom does process want to receive? */
  int p_sendto;

  struct proc *p_nextready;     /* pointer to next ready process */
  sigset_t p_pending;           /* bit map for pending signals */
  unsigned p_pendcount;         /* count of pending and unfinished signals */

  char p_name[16];              /* name of the process */
};

/* Guard word for task stacks. */
#define STACK_GUARD     ((reg_t) (sizeof(reg_t) == 2 ? 0xBEEF : 0xDEADBEEF))

/* Bits for p_flags in proc[].  A process is runnable iff p_flags == 0. */
#define P_SLOT_FREE      001    /* set when slot is not in use */
#define NO_MAP           002    /* keeps unmapped forked child from running */
#define SENDING          004    /* set when process blocked trying to send */
#define RECEIVING        010    /* set when process blocked trying to recv */
#define PENDING          020    /* set when inform() of signal pending */
#define SIG_PENDING      040    /* keeps to-be-signalled proc from running */
#define P_STOP          0100    /* set when process is being traced */

/* Magic process table addresses. */
#define BEG_PROC_ADDR (&proc[0])
#define END_PROC_ADDR (&proc[NR_TASKS + NR_PROCS])
#define END_TASK_ADDR (&proc[NR_TASKS])
#define BEG_SERV_ADDR (&proc[NR_TASKS])
#define BEG_USER_ADDR (&proc[NR_TASKS + LOW_USER])

#define NIL_PROC          ((struct proc *) 0)
```

```
#define isidlehardware(n) ((n) == IDLE || (n) == HARDWARE)
#define isokprocn(n)      ((unsigned) ((n) + NR_TASKS) < NR_PROCS + NR_TASKS)
#define isoksrc_dest(n)   (isokprocn(n) || (n) == ANY)
#define isoksusern(n)     ((unsigned) (n) < NR_PROCS)
#define isokusern(n)      ((unsigned) ((n) - LOW_USER) < NR_PROCS - LOW_USER)
#define isrxhardware(n)   ((n) == ANY || (n) == HARDWARE)
#define issysentn(n)      ((n) == FS_PROC_NR || (n) == MM_PROC_NR)
#define istaskp(p)        ((p) < END_TASK_ADDR && (p) != proc_addr(IDLE))
#define isuserp(p)        ((p) >= BEG_USER_ADDR)
#define proc_addr(n)      (pproc_addr + NR_TASKS)[(n)]
#define cproc_addr(n)     (&(proc + NR_TASKS)[(n)])
#define proc_number(p)    ((p)->p_nr)
#define proc_vir2phys(p, vir) \
                          (((phys_bytes)(p)->p_map[D].mem_phys << CLICK_SHIFT) \
                                                        + (vir_bytes) (vir))

EXTERN struct proc proc[NR_TASKS + NR_PROCS];   /* process table */
EXTERN struct proc *pproc_addr[NR_TASKS + NR_PROCS];
/* ptrs to process table slots; fast because now a process entry can be found
   by indexing the pproc_addr array, while accessing an element i requires
   a multiplication with sizeof(struct proc) to determine the address */
EXTERN struct proc *bill_ptr;   /* ptr to process to bill for clock ticks */
EXTERN struct proc *rdy_head[NQ];       /* pointers to ready list headers */
EXTERN struct proc *rdy_tail[NQ];       /* pointers to ready list tails */

#endif /* PROC_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/protect.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Constants for protected mode. */

/* Table sizes. */
#define GDT_SIZE (FIRST_LDT_INDEX + NR_TASKS + NR_PROCS) /* spec. and LDT's */
#define IDT_SIZE (IRQ8_VECTOR + 8)      /* only up to the highest vector */
#define LDT_SIZE         2      /* contains CS and DS only */

/* Fixed global descriptors.  1 to 7 are prescribed by the BIOS. */
#define GDT_INDEX        1      /* GDT descriptor */
#define IDT_INDEX        2      /* IDT descriptor */
#define DS_INDEX         3      /* kernel DS */
#define ES_INDEX         4      /* kernel ES (386: flag 4 Gb at startup) */
#define SS_INDEX         5      /* kernel SS (386: monitor SS at startup) */
#define CS_INDEX         6      /* kernel CS */
#define MON_CS_INDEX     7      /* temp for BIOS (386: monitor CS at startup) */
#define TSS_INDEX        8      /* kernel TSS */
#define DS_286_INDEX     9      /* scratch 16-bit source segment */
#define ES_286_INDEX    10      /* scratch 16-bit destination segment */
#define VIDEO_INDEX     11      /* video memory segment */
#define DP_ETH0_INDEX   12      /* Western Digital Etherplus buffer */
#define DP_ETH1_INDEX   13      /* Western Digital Etherplus buffer */
#define FIRST_LDT_INDEX 14      /* rest of descriptors are LDT's */

#define GDT_SELECTOR      0x08  /* (GDT_INDEX * DESC_SIZE) bad for asld */
#define IDT_SELECTOR      0x10  /* (IDT_INDEX * DESC_SIZE) */
```

```
#define DS_SELECTOR       0x18  /* (DS_INDEX * DESC_SIZE) */
#define ES_SELECTOR       0x20  /* (ES_INDEX * DESC_SIZE) */
#define FLAT_DS_SELECTOR  0x21  /* less privileged ES */
#define SS_SELECTOR       0x28  /* (SS_INDEX * DESC_SIZE) */
#define CS_SELECTOR       0x30  /* (CS_INDEX * DESC_SIZE) */
#define MON_CS_SELECTOR   0x38  /* (MON_CS_INDEX * DESC_SIZE) */
#define TSS_SELECTOR      0x40  /* (TSS_INDEX * DESC_SIZE) */
#define DS_286_SELECTOR   0x49  /* (DS_286_INDEX * DESC_SIZE + 1) */
#define ES_286_SELECTOR   0x51  /* (ES_286_INDEX * DESC_SIZE + 1) */
#define VIDEO_SELECTOR    0x59  /* (VIDEO_INDEX * DESC_SIZE + 1) */
#define DP_ETH0_SELECTOR  0x61  /* (DP_ETH0_INDEX * DESC_SIZE) */
#define DP_ETH1_SELECTOR  0x69  /* (DP_ETH1_INDEX * DESC_SIZE) */

/* Fixed local descriptors. */
#define CS_LDT_INDEX     0      /* process CS */
#define DS_LDT_INDEX     1      /* process DS=ES=FS=GS=SS */

/* Privileges. */
#define INTR_PRIVILEGE   0      /* kernel and interrupt handlers */
#define TASK_PRIVILEGE   1
#define USER_PRIVILEGE   3

/* 286 hardware constants. */

/* Exception vector numbers. */
#define BOUNDS_VECTOR       5   /* bounds check failed */
#define INVAL_OP_VECTOR     6   /* invalid opcode */
#define COPROC_NOT_VECTOR   7   /* coprocessor not available */
#define DOUBLE_FAULT_VECTOR 8
#define COPROC_SEG_VECTOR   9   /* coprocessor segment overrun */
#define INVAL_TSS_VECTOR   10   /* invalid TSS */
#define SEG_NOT_VECTOR     11   /* segment not present */
#define STACK_FAULT_VECTOR 12   /* stack exception */
#define PROTECTION_VECTOR  13   /* general protection */

/* Selector bits. */
#define TI            0x04      /* table indicator */
#define RPL           0x03      /* requester privilege level */

/* Descriptor structure offsets. */
#define DESC_BASE        2      /* to base_low */
#define DESC_BASE_MIDDLE 4      /* to base_middle */
#define DESC_ACCESS      5      /* to access byte */
#define DESC_SIZE        8      /* sizeof (struct segdesc_s) */

/* Segment sizes. */
#define MAX_286_SEG_SIZE 0x10000L

/* Base and limit sizes and shifts. */
#define BASE_MIDDLE_SHIFT   16  /* shift for base --> base_middle */

/* Access-byte and type-byte bits. */
#define PRESENT       0x80      /* set for descriptor present */
#define DPL           0x60      /* descriptor privilege level mask */
#define DPL_SHIFT        5
#define SEGMENT       0x10      /* set for segment-type descriptors */

/* Access-byte bits. */
#define EXECUTABLE    0x08      /* set for executable segment */
#define CONFORMING    0x04      /* set for conforming segment if executable */
```

```
#define EXPAND_DOWN   0x04      /* set for expand-down segment if !executable*/
#define READABLE      0x02      /* set for readable segment if executable */
#define WRITEABLE     0x02      /* set for writeable segment if !executable */
#define TSS_BUSY      0x02      /* set if TSS descriptor is busy */
#define ACCESSED      0x01      /* set if segment accessed */

/* Special descriptor types. */
#define AVL_286_TSS      1      /* available 286 TSS */
#define LDT              2      /* local descriptor table */
#define BUSY_286_TSS     3      /* set transparently to the software */
#define CALL_286_GATE    4      /* not used */
#define TASK_GATE        5      /* only used by debugger */
#define INT_286_GATE     6      /* interrupt gate, used for all vectors */
#define TRAP_286_GATE    7      /* not used */

/* Extra 386 hardware constants. */

/* Exception vector numbers. */
#define PAGE_FAULT_VECTOR   14
#define COPROC_ERR_VECTOR   16  /* coprocessor error */

/* Descriptor structure offsets. */
#define DESC_GRANULARITY     6  /* to granularity byte */
#define DESC_BASE_HIGH       7  /* to base_high */

/* Base and limit sizes and shifts. */
#define BASE_HIGH_SHIFT     24  /* shift for base --> base_high */
#define BYTE_GRAN_MAX   0xFFFFFL   /* maximum size for byte granular segment */
#define GRANULARITY_SHIFT   16  /* shift for limit --> granularity */
#define OFFSET_HIGH_SHIFT   16  /* shift for (gate) offset --> offset_high */
#define PAGE_GRAN_SHIFT     12  /* extra shift for page granular limits */

/* Type-byte bits. */
#define DESC_386_BIT      0x08  /* 386 types are obtained by ORing with this */
                                /* LDT's and TASK_GATE's don't need it */

/* Granularity byte. */
#define GRANULAR          0x80  /* set for 4K granularilty */
#define DEFAULT           0x40  /* set for 32-bit defaults (executable seg) */
#define BIG               0x40  /* set for "BIG" (expand-down seg) */
#define AVL               0x10  /* 0 for available */
#define LIMIT_HIGH        0x0F  /* mask for high bits of limit */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/sconst.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
! Miscellaneous constants used in assembler code.
W               =       _WORD_SIZE      ! Machine word size.

! Offsets in struct proc. They MUST match proc.h.
P_STACKBASE     =       0
#if _WORD_SIZE == 2
ESREG           =       P_STACKBASE
#else
GSREG           =       P_STACKBASE
FSREG           =       GSREG + 2       ! 386 introduces FS and GS segments
```

```
ESREG           =       FSREG + 2
#endif
DSREG           =       ESREG + 2
DIREG           =       DSREG + 2
SIREG           =       DIREG + W
BPREG           =       SIREG + W
STREG           =       BPREG + W       ! hole for another SP
BXREG           =       STREG + W
DXREG           =       BXREG + W
CXREG           =       DXREG + W
AXREG           =       CXREG + W
RETADR          =       AXREG + W       ! return address for save() call
PCREG           =       RETADR + W
CSREG           =       PCREG + W
PSWREG          =       CSREG + W
SPREG           =       PSWREG + W
SSREG           =       SPREG + W
P_STACKTOP      =       SSREG + W
P_LDT_SEL       =       P_STACKTOP
P_LDT           =       P_LDT_SEL + W

#if _WORD_SIZE == 2
Msize           =       12              ! size of a message in 16-bit words
#else
Msize           =       9               ! size of a message in 32-bit words
#endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/assert.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*
assert.h
*/
#ifndef ASSERT_H
#define ASSERT_H

#if DEBUG

#define INIT_ASSERT     static char *assert_file= __FILE__;

void bad_assertion(char *file, int line, char *what);
void bad_compare(char *file, int line, int lhs, char *what, int rhs);

#define assert(x)       (!(x) ? bad_assertion(assert_file, __LINE__, #x) \
                                                                : (void) 0)
#define compare(a,t,b)  (!((a) t (b)) ? bad_compare(assert_file, __LINE__, \
                                (a), #a " " #t " " #b, (b)) : (void) 0)
#else /* !DEBUG */

#define INIT_ASSERT     /* nothing */

#define assert(x)       (void)0
#define compare(a,t,b)  (void)0

#endif /* !DEBUG */
```

```
#endif /* ASSERT_H */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The object file of "table.c" contains all the data.  In the *.h files,
 * declared variables appear with EXTERN in front of them, as in
 *
 *    EXTERN int x;
 *
 * Normally EXTERN is defined as extern, so when they are included in another
 * file, no storage is allocated.  If the EXTERN were not present, but just
 * say,
 *
 *    int x;
 *
 * then including this file in several source files would cause 'x' to be
 * declared several times.  While some linkers accept this, others do not,
 * so they are declared extern when included normally.  However, it must
 * be declared for real somewhere.  That is done here, by redefining
 * EXTERN as the null string, so the inclusion of all the *.h files in
 * table.c actually generates storage for them.  All the initialized
 * variables are also declared here, since
 *
 * extern int x = 4;
 *
 * is not allowed.  If such variables are shared, they must also be declared
 * in one of the *.h files without the initialization.
 */

#define _TABLE

#include "kernel.h"
#include <termios.h>
#include <minix/com.h>
#include "proc.h"
#include "tty.h"

/* The startup routine of each task is given below, from -NR_TASKS upwards.
 * The order of the names here MUST agree with the numerical values assigned to
 * the tasks in <minix/com.h>.
 */
#define SMALL_STACK     (128 * sizeof(char *))

#define TTY_STACK       (3 * SMALL_STACK)
#define SYN_ALRM_STACK  SMALL_STACK

#define DP8390_STACK    (SMALL_STACK * ENABLE_NETWORKING)

#if (CHIP == INTEL)
#define IDLE_STACK      ((3+3+4) * sizeof(char *))  /* 3 intr, 3 temps, 4 db */
#else
#define IDLE_STACK      SMALL_STACK
#endif
```

```
#define PRINTER_STACK   SMALL_STACK

#if (CHIP == INTEL)
#define WINCH_STACK     (2 * SMALL_STACK * ENABLE_WINI)
#else
#define WINCH_STACK     (3 * SMALL_STACK * ENABLE_WINI)
#endif

#if (MACHINE == ATARI)
#define SCSI_STACK      (3 * SMALL_STACK)
#endif

#if (MACHINE == IBM_PC)
#define SCSI_STACK      (2 * SMALL_STACK * ENABLE_SCSI)
#endif

#define CDROM_STACK     (4 * SMALL_STACK * ENABLE_CDROM)
#define AUDIO_STACK     (4 * SMALL_STACK * ENABLE_AUDIO)
#define MIXER_STACK     (4 * SMALL_STACK * ENABLE_AUDIO)

#define FLOP_STACK      (3 * SMALL_STACK)
#define MEM_STACK       SMALL_STACK
#define CLOCK_STACK     SMALL_STACK
#define SYS_STACK       SMALL_STACK
#define HARDWARE_STACK  0               /* dummy task, uses kernel stack */


#define TOT_STACK_SPACE         (TTY_STACK + DP8390_STACK + SCSI_STACK + \
        SYN_ALRM_STACK + IDLE_STACK + HARDWARE_STACK + PRINTER_STACK + \
        WINCH_STACK + FLOP_STACK + MEM_STACK + CLOCK_STACK + SYS_STACK + \
        CDROM_STACK + AUDIO_STACK + MIXER_STACK)


/* SCSI, CDROM and AUDIO may in the future have different choices like
 * WINCHESTER, but for now the choice is fixed.
 */
#define scsi_task       aha_scsi_task
#define cdrom_task      mcd_task
#define audio_task      dsp_task


/*
 * Some notes about the following table:
 *  1) The tty_task should always be first so that other tasks can use printf
 *     if their initialisation has problems.
 *  2) If you add a new kernel task, add it before the printer task.
 *  3) The task name is used for the process name (p_name).
 */

PUBLIC struct tasktab tasktab[] = {
        { tty_task,             TTY_STACK,      "TTY"           },
#if ENABLE_NETWORKING
        { dp8390_task,          DP8390_STACK,   "DP8390"        },
#endif
#if ENABLE_CDROM
        { cdrom_task,           CDROM_STACK,    "CDROM"         },
#endif
#if ENABLE_AUDIO
        { audio_task,           AUDIO_STACK,    "AUDIO"         },
        { mixer_task,           MIXER_STACK,    "MIXER"         },
```

```
#endif
#if ENABLE_SCSI
        { scsi_task,            SCSI_STACK,     "SCSI"          },
#endif
#if ENABLE_WINI
        { winchester_task,      WINCH_STACK,    "WINCH"         },
#endif
        { syn_alrm_task,        SYN_ALRM_STACK, "SYN_AL"        },
        { idle_task,            IDLE_STACK,     "IDLE"          },
        { printer_task,         PRINTER_STACK,  "PRINTER"       },
        { floppy_task,          FLOP_STACK,     "FLOPPY"        },
        { mem_task,             MEM_STACK,      "MEMORY"        },
        { clock_task,           CLOCK_STACK,    "CLOCK"         },
        { sys_task,             SYS_STACK,      "SYS"           },
        { 0,                    HARDWARE_STACK, "HARDWAR"       },
        { 0,                    0,              "MM"            },
        { 0,                    0,              "FS"            },
#if ENABLE_NETWORKING
        { 0,                    0,              "INET"          },
#endif
        { 0,                    0,              "INIT"          },
};

/* Stack space for all the task stacks.  (Declared as (char *) to align it.) */
PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];

/*
 * The number of kernel tasks must be the same as NR_TASKS.
 * If NR_TASKS is not correct then you will get the compile error:
 *   "array size is negative"
 */

#define NKT (sizeof tasktab / sizeof (struct tasktab) - (INIT_PROC_NR + 1))

extern int dummy_tasktab_check[NR_TASKS == NKT ? 1 : -1];
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/mpx.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#
! Chooses between the 8086 and 386 versions of the Minix startup code.

#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/mpx386.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

#
! This file contains the assembler startup code for Minix and the 32-bit
! interrupt handlers.  It cooperates with start.c to set up a good
! environment for main().

! This file is part of the lowest layer of the MINIX kernel.  The other part
! is "proc.c".  The lowest layer does process switching and message handling.

! Every transition to the kernel goes through this file.  Transitions are
! caused by sending/receiving messages and by most interrupts.  (RS232
! interrupts may be handled in the file "rs2.s" and then they rarely enter
! the kernel.)

! Transitions to the kernel may be nested.  The initial entry may be with a
! system call, exception or hardware interrupt; reentries may only be made
! by hardware interrupts.  The count of reentries is kept in "k_reenter".
! It is important for deciding whether to switch to the kernel stack and
! for protecting the message passing code in "proc.c".

! For the message passing trap, most of the machine state is saved in the
! proc table.  (Some of the registers need not be saved.)  Then the stack is
! switched to "k_stack", and interrupts are reenabled.  Finally, the system
! call handler (in C) is called.  When it returns, interrupts are disabled
! again and the code falls into the restart routine, to finish off held-up
! interrupts and run the process or task whose pointer is in "proc_ptr".

! Hardware interrupt handlers do the same, except  (1) The entire state must
! be saved.  (2) There are too many handlers to do this inline, so the save
! routine is called.  A few cycles are saved by pushing the address of the
! appropiate restart routine for a return later.  (3) A stack switch is
! avoided when the stack is already switched.  (4) The (master) 8259 interrupt
! controller is reenabled centrally in save().  (5) Each interrupt handler
! masks its interrupt line using the 8259 before enabling (other unmasked)
! interrupts, and unmasks it after servicing the interrupt.  This limits the
! nest level to the number of lines and protects the handler from itself.

! For communication with the boot monitor at startup time some constant
! data are compiled into the beginning of the text segment. This facilitates
! reading the data at the start of the boot process, since only the first
! sector of the file needs to be read.

! Some data storage is also allocated at the end of this file. This data
! will be at the start of the data segment of the kernel and will be read
! and modified by the boot monitor before the kernel starts.

! sections

.sect .text
begtext:
.sect .rom
begrom:
.sect .data
begdata:
.sect .bss
begbss:
```

```
#include <minix/config.h>
#include <minix/const.h>
#include <minix/com.h>
#include "const.h"
#include "protect.h"
#include "sconst.h"

/* Selected 386 tss offsets. */
#define TSS3_S_SP0      4

! Exported functions
! Note: in assembly language the .define statement applied to a function name
! is loosely equivalent to a prototype in C code -- it makes it possible to
! link to an entity declared in the assembly code but does not create
! the entity.

.define _idle_task
.define _restart
.define save

.define _divide_error
.define _single_step_exception
.define _nmi
.define _breakpoint_exception
.define _overflow
.define _bounds_check
.define _inval_opcode
.define _copr_not_available
.define _double_fault
.define _copr_seg_overrun
.define _inval_tss
.define _segment_not_present
.define _stack_exception
.define _general_protection
.define _page_fault
.define _copr_error

.define _hwint00        ! handlers for hardware interrupts
.define _hwint01
.define _hwint02
.define _hwint03
.define _hwint04
.define _hwint05
.define _hwint06
.define _hwint07
.define _hwint08
.define _hwint09
.define _hwint10
.define _hwint11
.define _hwint12
.define _hwint13
.define _hwint14
.define _hwint15

.define _s_call
.define _p_s_call
.define _level0_call

! Imported functions.
```

```

.extern _cstart
.extern _main
.extern _exception
.extern _interrupt
.extern _sys_call
.extern _unhold

! Exported variables.
! Note: when used with a variable the .define does not reserve storage,
! it makes the name externally visible so it may be linked to.

.define begbss
.define begdata
.define _sizes

! Imported variables.

.extern _gdt
.extern _code_base
.extern _data_base
.extern _held_head
.extern _k_reenter
.extern _pc_at
.extern _proc_ptr
.extern _ps_mca
.extern _tss
.extern _level0_func
.extern _mon_sp
.extern _mon_return
.extern _reboot_code

.sect .text
!*===========================================================================*
!*                              MINIX                                        *
!*===========================================================================*
MINIX:                          ! this is the entry point for the MINIX kernel
        jmp     over_flags      ! skip over the next few bytes
        .data2  CLICK_SHIFT     ! for the monitor: memory granularity
flags:
        .data2  0x002D          ! boot monitor flags:
                                !       call in 386 mode, make stack,
                                !       load high, will return
        nop                     ! extra byte to sync up disassembler
over_flags:

! Set up a C stack frame on the monitor stack.  (The monitor sets cs and ds
! right.  The ss descriptor still references the monitor data segment.)
        movzx   esp, sp         ! monitor stack is a 16 bit stack
        push    ebp
        mov     ebp, esp
        push    esi
        push    edi
        cmp     4(ebp), 0       ! nonzero if return possible
        jz      noret
        inc     (_mon_return)
noret:  mov     (_mon_sp), esp  ! save stack pointer for later return

! Copy the monitor global descriptor table to the address space of kernel and
! switch over to it.  Prot_init() can then update it with immediate effect.
```

```

        sgdt    (_gdt+GDT_SELECTOR)             ! get the monitor gdtr
        mov     esi, (_gdt+GDT_SELECTOR+2)      ! absolute address of GDT
        mov     ebx, _gdt                       ! address of kernel GDT
        mov     ecx, 8*8                        ! copying eight descriptors
copygdt:
 eseg   movb    al, (esi)
        movb    (ebx), al
        inc     esi
        inc     ebx
        loop    copygdt
        mov     eax, (_gdt+DS_SELECTOR+2)       ! base of kernel data
        and     eax, 0x00FFFFFF                 ! only 24 bits
        add     eax, _gdt                       ! eax = vir2phys(gdt)
        mov     (_gdt+GDT_SELECTOR+2), eax      ! set base of GDT
        lgdt    (_gdt+GDT_SELECTOR)             ! switch over to kernel GDT

! Locate boot parameters, set up kernel segment registers and stack.
        mov     ebx, 8(ebp)     ! boot parameters offset
        mov     edx, 12(ebp)    ! boot parameters length
        mov     ax, ds          ! kernel data
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        mov     ss, ax
        mov     esp, k_stktop   ! set sp to point to the top of kernel stack

! Call C startup code to set up a proper environment to run main().
        push    edx
        push    ebx
        push    SS_SELECTOR
        push    MON_CS_SELECTOR
        push    DS_SELECTOR
        push    CS_SELECTOR
        call    _cstart         ! cstart(cs, ds, mcs, mds, parmoff, parmlen)
        add     esp, 6*4

! Reload gdtr, idtr and the segment registers to global descriptor table set
! up by prot_init().

        lgdt    (_gdt+GDT_SELECTOR)
        lidt    (_gdt+IDT_SELECTOR)

        jmpf    CS_SELECTOR:csinit
csinit:
    o16 mov     ax, DS_SELECTOR
        mov     ds, ax
        mov     es, ax
        mov     fs, ax
        mov     gs, ax
        mov     ss, ax
    o16 mov     ax, TSS_SELECTOR        ! no other TSS is used
        ltr     ax
        push    0                       ! set flags to known good state
        popf                            ! esp, clear nested task and int enable

        jmp     _main                   ! main()


!*===========================================================================*
```

```
!*                              interrupt handlers                              *
!*              interrupt handlers for 386 32-bit protected mode                *
!*==============================================================================*

!*==============================================================================*
!*                              hwint00 - 07                                    *
!*==============================================================================*
! Note this is a macro, it looks like a subroutine.
#define hwint_master(irq)       \
        call    save                    /* save interrupted process state */;\
        inb     INT_CTLMASK                                                 ;\
        orb     al, [1<<irq]                                                ;\
        outb    INT_CTLMASK             /* disable the irq                */;\
        movb    al, ENABLE                                                  ;\
        outb    INT_CTL                 /* reenable master 8259           */;\
        sti                             /* enable interrupts              */;\
        push    irq                     /* irq                            */;\
        call    (_irq_table + 4*irq)    /* eax = (*irq_table[irq])(irq)   */;\
        pop     ecx                                                         ;\
        cli                             /* disable interrupts             */;\
        test    eax, eax                /* need to reenable irq?          */;\
        jz      0f                                                          ;\
        inb     INT_CTLMASK                                                 ;\
        andb    al, ~[1<<irq]                                               ;\
        outb    INT_CTLMASK             /* enable the irq                 */;\
0:      ret                             /* restart (another) process      */

! Each of these entry points is an expansion of the hwint_master macro
        .align  16
_hwint00:               ! Interrupt routine for irq 0 (the clock).
        hwint_master(0)

        .align  16
_hwint01:               ! Interrupt routine for irq 1 (keyboard)
        hwint_master(1)

        .align  16
_hwint02:               ! Interrupt routine for irq 2 (cascade!)
        hwint_master(2)

        .align  16
_hwint03:               ! Interrupt routine for irq 3 (second serial)
        hwint_master(3)

        .align  16
_hwint04:               ! Interrupt routine for irq 4 (first serial)
        hwint_master(4)

        .align  16
_hwint05:               ! Interrupt routine for irq 5 (XT winchester)
        hwint_master(5)

        .align  16
_hwint06:               ! Interrupt routine for irq 6 (floppy)
        hwint_master(6)

        .align  16
_hwint07:               ! Interrupt routine for irq 7 (printer)
        hwint_master(7)
```

```
!*===========================================================================*
!*                              hwint08 - 15                                 *
!*===========================================================================*
! Note this is a macro, it looks like a subroutine.
#define hwint_slave(irq)        \
        call    save                    /* save interrupted process state */;\
        inb     INT2_CTLMASK                                                ;\
        orb     al, [1<<[irq-8]]                                            ;\
        outb    INT2_CTLMASK            /* disable the irq                */;\
        movb    al, ENABLE                                                  ;\
        outb    INT_CTL                 /* reenable master 8259           */;\
        jmp     .+2                     /* delay                          */;\
        outb    INT2_CTL                /* reenable slave 8259            */;\
        sti                             /* enable interrupts              */;\
        push    irq                     /* irq                            */;\
        call    (_irq_table + 4*irq)    /* eax = (*irq_table[irq])(irq)   */;\
        pop     ecx                                                         ;\
        cli                             /* disable interrupts             */;\
        test    eax, eax                /* need to reenable irq?          */;\
        jz      0f                                                          ;\
        inb     INT2_CTLMASK                                                ;\
        andb    al, ~[1<<[irq-8]]                                           ;\
        outb    INT2_CTLMASK            /* enable the irq                 */;\
0:      ret                             /* restart (another) process      */

! Each of these entry points is an expansion of the hwint_slave macro
        .align  16
_hwint08:               ! Interrupt routine for irq 8 (realtime clock)
        hwint_slave(8)

        .align  16
_hwint09:               ! Interrupt routine for irq 9 (irq 2 redirected)
        hwint_slave(9)

        .align  16
_hwint10:               ! Interrupt routine for irq 10
        hwint_slave(10)

        .align  16
_hwint11:               ! Interrupt routine for irq 11
        hwint_slave(11)

        .align  16
_hwint12:               ! Interrupt routine for irq 12
        hwint_slave(12)

        .align  16
_hwint13:               ! Interrupt routine for irq 13 (FPU exception)
        hwint_slave(13)

        .align  16
_hwint14:               ! Interrupt routine for irq 14 (AT winchester)
        hwint_slave(14)

        .align  16
_hwint15:               ! Interrupt routine for irq 15
        hwint_slave(15)

!*===========================================================================*
!*                              save                                         *
```

```
!*===========================================================================*
! Save for protected mode.
! This is much simpler than for 8086 mode, because the stack already points
! into the process table, or has already been switched to the kernel stack.

        .align  16
save:
        cld                     ! set direction flag to a known value
        pushad                  ! save "general" registers
    o16 push    ds              ! save ds
    o16 push    es              ! save es
    o16 push    fs              ! save fs
    o16 push    gs              ! save gs
        mov     dx, ss          ! ss is kernel data segment
        mov     ds, dx          ! load rest of kernel segments
        mov     es, dx          ! kernel does not use fs, gs
        mov     eax, esp        ! prepare to return
        incb    (_k_reenter)    ! from -1 if not reentering
        jnz     set_restart1    ! stack is already kernel stack
        mov     esp, k_stktop
        push    _restart        ! build return address for int handler
        xor     ebp, ebp        ! for stacktrace
        jmp     RETADR-P_STACKBASE(eax)

        .align  4
set_restart1:
        push    restart1
        jmp     RETADR-P_STACKBASE(eax)

!*===========================================================================*
!*                              _s_call                                      *
!*===========================================================================*
        .align  16
_s_call:
_p_s_call:
        cld                     ! set direction flag to a known value
        sub     esp, 6*4        ! skip RETADR, eax, ecx, edx, ebx, est
        push    ebp             ! stack already points into proc table
        push    esi
        push    edi
    o16 push    ds
    o16 push    es
    o16 push    fs
    o16 push    gs
        mov     dx, ss
        mov     ds, dx
        mov     es, dx
        incb    (_k_reenter)
        mov     esi, esp        ! assumes P_STACKBASE == 0
        mov     esp, k_stktop
        xor     ebp, ebp        ! for stacktrace
                                ! end of inline save
        sti                     ! allow SWITCHER to be interrupted
                                ! now set up parameters for sys_call()
        push    ebx             ! pointer to user message
        push    eax             ! src/dest
        push    ecx             ! SEND/RECEIVE/BOTH
        call    _sys_call       ! sys_call(function, src_dest, m_ptr)
                                ! caller is now explicitly in proc_ptr
        mov     AXREG(esi), eax ! sys_call MUST PRESERVE si
```

```
        cli                     ! disable interrupts

! Fall into code to restart proc/task running.

!*===========================================================================*
!*                              restart                                      *
!*===========================================================================*
_restart:

! Flush any held-up interrupts.
! This reenables interrupts, so the current interrupt handler may reenter.
! This does not matter, because the current handler is about to exit and no
! other handlers can reenter since flushing is only done when k_reenter == 0.

        cmp     (_held_head), 0 ! do fast test to usually avoid function call
        jz      over_call_unhold
        call    _unhold         ! this is rare so overhead acceptable
over_call_unhold:
        mov     esp, (_proc_ptr)        ! will assume P_STACKBASE == 0
        lldt    P_LDT_SEL(esp)          ! enable segment descriptors for task
        lea     eax, P_STACKTOP(esp)    ! arrange for next interrupt
        mov     (_tss+TSS3_S_SP0), eax  ! to save state in process table
restart1:
        decb    (_k_reenter)
    o16 pop     gs
    o16 pop     fs
    o16 pop     es
    o16 pop     ds
        popad
        add     esp, 4          ! skip return adr
        iretd                   ! continue process

!*===========================================================================*
!*                              exception handlers                           *
!*===========================================================================*
_divide_error:
        push    DIVIDE_VECTOR
        jmp     exception

_single_step_exception:
        push    DEBUG_VECTOR
        jmp     exception

_nmi:
        push    NMI_VECTOR
        jmp     exception

_breakpoint_exception:
        push    BREAKPOINT_VECTOR
        jmp     exception

_overflow:
        push    OVERFLOW_VECTOR
        jmp     exception

_bounds_check:
        push    BOUNDS_VECTOR
        jmp     exception

_inval_opcode:
```

```
        push    INVAL_OP_VECTOR
        jmp     exception

_copr_not_available:
        push    COPROC_NOT_VECTOR
        jmp     exception

_double_fault:
        push    DOUBLE_FAULT_VECTOR
        jmp     errexception

_copr_seg_overrun:
        push    COPROC_SEG_VECTOR
        jmp     exception

_inval_tss:
        push    INVAL_TSS_VECTOR
        jmp     errexception

_segment_not_present:
        push    SEG_NOT_VECTOR
        jmp     errexception

_stack_exception:
        push    STACK_FAULT_VECTOR
        jmp     errexception

_general_protection:
        push    PROTECTION_VECTOR
        jmp     errexception

_page_fault:
        push    PAGE_FAULT_VECTOR
        jmp     errexception

_copr_error:
        push    COPROC_ERR_VECTOR
        jmp     exception

!*===========================================================================*
!*                              exception                                    *
!*===========================================================================*
! This is called for all exceptions which do not push an error code.

        .align  16
exception:
 sseg   mov     (trap_errno), 0         ! clear trap_errno
 sseg   pop     (ex_number)
        jmp     exception1

!*===========================================================================*
!*                              errexception                                 *
!*===========================================================================*
! This is called for all exceptions which push an error code.

        .align  16
errexception:
 sseg   pop     (ex_number)
 sseg   pop     (trap_errno)
exception1:                             ! Common for all exceptions.
```

```
        push    eax                     ! eax is scratch register
        mov     eax, 0+4(esp)           ! old eip
 sseg   mov     (old_eip), eax
        movzx   eax, 4+4(esp)           ! old cs
 sseg   mov     (old_cs), eax
        mov     eax, 8+4(esp)           ! old eflags
 sseg   mov     (old_eflags), eax
        pop     eax
        call    save
        push    (old_eflags)
        push    (old_cs)
        push    (old_eip)
        push    (trap_errno)
        push    (ex_number)
        call    _exception              ! (ex_number, trap_errno, old_eip,
                                        !       old_cs, old_eflags)
        add     esp, 5*4
        cli
        ret

!*===========================================================================*
!*                              level0_call                                  *
!*===========================================================================*
_level0_call:
        call    save
        jmp     (_level0_func)

!*===========================================================================*
!*                              idle_task                                    *
!*===========================================================================*
_idle_task:                     ! executed when there is no work
        jmp     _idle_task      ! a "hlt" before this fails in protected mode

!*===========================================================================*
!*                              data                                         *
!*===========================================================================*
! These declarations assure that storage will be allocated at the very
! beginning of the kernel data section, so the boot monitor can be easily
! told how to patch these locations. Note that the magic number is put
! here by the compiler, but will be read by, and then overwritten by,
! the boot monitor. When the kernel starts the sizes array will be
! found here, as if it had been initialized by the compiler.

.sect .rom      ! Before the string table please
_sizes:                         ! sizes of kernel, mm, fs filled in by boot
        .data2  0x526F          ! this must be the first data entry (magic #)
        .space  16*2*2-2        ! monitor uses previous word and this space
                                ! extra space allows for additional servers
.sect .bss
k_stack:
        .space  K_STACK_BYTES   ! kernel stack
k_stktop:                       ! top of kernel stack
        .comm   ex_number, 4
        .comm   trap_errno, 4
        .comm   old_eip, 4
        .comm   old_cs, 4
        .comm   old_eflags, 4
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  src/kernel/start.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the C startup code for Minix on Intel processors.
 * It cooperates with mpx.s to set up a good environment for main().
 *
 * This code runs in real mode for a 16 bit kernel and may have to switch
 * to protected mode for a 286.
 *
 * For a 32 bit kernel this already runs in protected mode, but the selectors
 * are still those given by the BIOS with interrupts disabled, so the
 * descriptors need to be reloaded and interrupt descriptors made.
 */

#include "kernel.h"
#include <stdlib.h>
#include <minix/boot.h>
#include "protect.h"

PRIVATE char k_environ[256];    /* environment strings passed by loader */

FORWARD _PROTOTYPE( int k_atoi, (char *s) );


/*===========================================================================*
 *                              cstart                                       *
 *===========================================================================*/
PUBLIC void cstart(cs, ds, mcs, mds, parmoff, parmsize)
U16_t cs, ds;                   /* Kernel code and data segment */
U16_t mcs, mds;                 /* Monitor code and data segment */
U16_t parmoff, parmsize;        /* boot parameters offset and length */
{
/* Perform system initializations prior to calling main(). */

  register char *envp;
  phys_bytes mcode_base, mdata_base;
  unsigned mon_start;

  /* Record where the kernel and the monitor are. */
  code_base = seg2phys(cs);
  data_base = seg2phys(ds);
  mcode_base = seg2phys(mcs);
  mdata_base = seg2phys(mds);

  /* Initialize protected mode descriptors. */
  prot_init();

  /* Copy the boot parameters to kernel memory. */
  if (parmsize > sizeof k_environ - 2) parmsize = sizeof k_environ - 2;
  phys_copy(mdata_base + parmoff, vir2phys(k_environ), (phys_bytes) parmsize);

  /* Convert important boot environment variables. */
  boot_parameters.bp_rootdev = k_atoi(k_getenv("rootdev"));
  boot_parameters.bp_ramimagedev = k_atoi(k_getenv("ramimagedev"));
  boot_parameters.bp_ramsize = k_atoi(k_getenv("ramsize"));
  boot_parameters.bp_processor = k_atoi(k_getenv("processor"));

  /* Type of VDU: */
```

```
  envp = k_getenv("video");
  if (strcmp(envp, "ega") == 0) ega = TRUE;
  if (strcmp(envp, "vga") == 0) vga = ega = TRUE;

  /* Memory sizes: */
  low_memsize = k_atoi(k_getenv("memsize"));
  ext_memsize = k_atoi(k_getenv("emssize"));

  /* Processor? */
  processor = boot_parameters.bp_processor;     /* 86, 186, 286, 386, ... */

  /* XT, AT or MCA bus? */
  envp = k_getenv("bus");
  if (envp == NIL_PTR || strcmp(envp, "at") == 0) {
        pc_at = TRUE;
  } else
  if (strcmp(envp, "mca") == 0) {
        pc_at = ps_mca = TRUE;
  }

  /* Decide if mode is protected. */
#if _WORD_SIZE == 2
  protected_mode = processor >= 286;
#endif

  /* Is there a monitor to return to?  If so then keep it safe. */
  if (!protected_mode) mon_return = 0;
  mon_start = mcode_base / 1024;
  if (mon_return && low_memsize > mon_start) low_memsize = mon_start;

  /* Return to assembler code to switch to protected mode (if 286), reload
   * selectors and call main().
   */
}


/*===========================================================================*
 *                              k_atoi                                       *
 *===========================================================================*/
PRIVATE int k_atoi(s)
register char *s;
{
/* Convert string to integer. */

  return strtol(s, (char **) NULL, 10);
}


/*===========================================================================*
 *                              k_getenv                                     *
 *===========================================================================*/
PUBLIC char *k_getenv(name)
char *name;
{
/* Get environment value - kernel version of getenv to avoid setting up the
 * usual environment array.
 */

  register char *namep;
  register char *envp;
```

```
  for (envp = k_environ; *envp != 0;) {
        for (namep = name; *namep != 0 && *namep == *envp; namep++, envp++)
                ;
        if (*namep == '\0' && *envp == '=') return(envp + 1);
        while (*envp++ != 0)
                ;
  }
  return(NIL_PTR);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the main program of MINIX.  The routine main()
 * initializes the system and starts the ball rolling by setting up the proc
 * table, interrupt vectors, and scheduling each task to run to initialize
 * itself.
 *
 * The entries into this file are:
 *   main:              MINIX main program
 *   panic:             abort MINIX due to a fatal error
 */

#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"


/*===========================================================================*
 *                                    main                                   *
 *===========================================================================*/
PUBLIC void main()
{
/* Start the ball rolling. */

  register struct proc *rp;
  register int t;
  int sizeindex;
  phys_clicks text_base;
  vir_clicks text_clicks;
  vir_clicks data_clicks;
  phys_bytes phys_b;
  reg_t ktsb;                   /* kernel task stack base */
  struct memory *memp;
  struct tasktab *ttp;

  /* Initialize the interrupt controller. */
  intr_init(1);

  /* Interpret memory sizes. */
```

```
        mem_init();

        /* Clear the process table.
         * Set up mappings for proc_addr() and proc_number() macros.
         */
        for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
                rp->p_flags = P_SLOT_FREE;
                rp->p_nr = t;           /* proc number from ptr */
                (pproc_addr + NR_TASKS)[t] = rp;        /* proc ptr from number */
        }

        /* Set up proc table entries for tasks and servers.  The stacks of the
         * kernel tasks are initialized to an array in data space.  The stacks
         * of the servers have been added to the data segment by the monitor, so
         * the stack pointer is set to the end of the data segment.  All the
         * processes are in low memory on the 8086.  On the 386 only the kernel
         * is in low memory, the rest if loaded in extended memory.
         */

        /* Task stacks. */
        ktsb = (reg_t) t_stack;

        for (t = -NR_TASKS; t <= LOW_USER; ++t) {
                rp = proc_addr(t);                      /* t's process slot */
                ttp = &tasktab[t + NR_TASKS];           /* t's task attributes */
                strcpy(rp->p_name, ttp->name);
                if (t < 0) {
                        if (ttp->stksize > 0) {
                                rp->p_stguard = (reg_t *) ktsb;
                                *rp->p_stguard = STACK_GUARD;
                        }
                        ktsb += ttp->stksize;
                        rp->p_reg.sp = ktsb;
                        text_base = code_base >> CLICK_SHIFT;
                                                /* tasks are all in the kernel */
                        sizeindex = 0;          /* and use the full kernel sizes */
                        memp = &mem[0];         /* remove from this memory chunk */
                } else {
                        sizeindex = 2 * t + 2;  /* MM, FS, INIT have their own sizes */
                }
                rp->p_reg.pc = (reg_t) ttp->initial_pc;
                rp->p_reg.psw = istaskp(rp) ? INIT_TASK_PSW : INIT_PSW;

                text_clicks = sizes[sizeindex];
                data_clicks = sizes[sizeindex + 1];
                rp->p_map[T].mem_phys = text_base;
                rp->p_map[T].mem_len  = text_clicks;
                rp->p_map[D].mem_phys = text_base + text_clicks;
                rp->p_map[D].mem_len  = data_clicks;
                rp->p_map[S].mem_phys = text_base + text_clicks + data_clicks;
                rp->p_map[S].mem_vir  = data_clicks;    /* empty - stack is in data */
                text_base += text_clicks + data_clicks; /* ready for next, if server */
                memp->size -= (text_base - memp->base);
                memp->base = text_base;                 /* memory no longer free */

                if (t >= 0) {
                        /* Initialize the server stack pointer.  Take it down one word
                         * to give crtso.s something to use as "argc".
                         */
                        rp->p_reg.sp = (rp->p_map[S].mem_vir +
```

```
                                rp->p_map[S].mem_len) << CLICK_SHIFT;
                rp->p_reg.sp -= sizeof(reg_t);
        }

#if _WORD_SIZE == 4
        /* Servers are loaded in extended memory if in 386 mode. */
        if (t < 0) {
                memp = &mem[1];
                text_base = 0x100000 >> CLICK_SHIFT;
        }
#endif
        if (!isidlehardware(t)) lock_ready(rp); /* IDLE, HARDWARE neveready */
        rp->p_flags = 0;

        alloc_segments(rp);
  }

  proc[NR_TASKS+INIT_PROC_NR].p_pid = 1;/* INIT of course has pid 1 */
  bill_ptr = proc_addr(IDLE);           /* it has to point somewhere */
  lock_pick_proc();

  /* Now go to the assembly code to start running the current process. */
  restart();
}


/*===========================================================================*
 *                                   panic                                   *
 *===========================================================================*/
PUBLIC void panic(s,n)
_CONST char *s;
int n;
{
/* The system has run aground of a fatal error.  Terminate execution.
 * If the panic originated in MM or FS, the string will be empty and the
 * file system already syncked.  If the panic originates in the kernel, we are
 * kind of stuck.
 */

  if (*s != 0) {
        printf("\nKernel panic: %s",s);
        if (n != NO_NUM) printf(" %d", n);
        printf("\n");
  }
  wreboot(RBT_PANIC);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/proc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains essentially all of the process and message handling.
 * It has two main entry points from the outside:
 *
 *   sys_call:   called when a process or task does SEND, RECEIVE or SENDREC
 *   interrupt: called by interrupt routines to send a message to task
 *
 * It also has several minor entry points:
 *
 *   lock_ready:      put a process on one of the ready queues so it can be run
 *   lock_unready:    remove a process from the ready queues
 *   lock_sched:      a process has run too long; schedule another one
 *   lock_mini_send:  send a message (used by interrupt signals, etc.)
 *   lock_pick_proc:  pick a process to run (used by system initialization)
 *   unhold:          repeat all held-up interrupts
 */

#include "kernel.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

PRIVATE unsigned char switching;        /* nonzero to inhibit interrupt() */

FORWARD _PROTOTYPE( int mini_send, (struct proc *caller_ptr, int dest,
                message *m_ptr) );
FORWARD _PROTOTYPE( int mini_rec, (struct proc *caller_ptr, int src,
                message *m_ptr) );
FORWARD _PROTOTYPE( void ready, (struct proc *rp) );
FORWARD _PROTOTYPE( void sched, (void) );
FORWARD _PROTOTYPE( void unready, (struct proc *rp) );
FORWARD _PROTOTYPE( void pick_proc, (void) );

#define CopyMess(s,sp,sm,dp,dm) \
        cp_mess(s, (sp)->p_map[D].mem_phys, (vir_bytes)sm, (dp)->p_map[D].mem_phys, (vir_bytes)dm)

/*===========================================================================*
 *                              interrupt                                    *
 *===========================================================================*/
PUBLIC void interrupt(task)
int task;                       /* number of task to be started */
{
/* An interrupt has occurred.  Schedule the task that handles it. */

  register struct proc *rp;     /* pointer to task's proc entry */

  rp = proc_addr(task);

  /* If this call would compete with other process-switching functions, put
   * it on the 'held' queue to be flushed at the next non-competing restart().
   * The competing conditions are:
   * (1) k_reenter == (typeof k_reenter) -1:
   *     Call from the task level, typically from an output interrupt
   *     routine.  An interrupt handler might reenter interrupt().  Rare,
   *     so not worth special treatment.
   * (2) k_reenter > 0:
```

```
   *     Call from a nested interrupt handler.  A previous interrupt handler
   *     might be inside interrupt() or sys_call().
   * (3) switching != 0:
   *     Some process-switching function other than interrupt() is being
   *     called from the task level, typically sched() from CLOCK.  An
   *     interrupt handler might call interrupt and pass the k_reenter test.
   */
  if (k_reenter != 0 || switching) {
        lock();
        if (!rp->p_int_held) {
                rp->p_int_held = TRUE;
                if (held_head != NIL_PROC)
                        held_tail->p_nextheld = rp;
                else
                        held_head = rp;
                held_tail = rp;
                rp->p_nextheld = NIL_PROC;
        }
        unlock();
        return;
  }

  /* If task is not waiting for an interrupt, record the blockage. */
  if ( (rp->p_flags & (RECEIVING | SENDING)) != RECEIVING ||
      !isrxhardware(rp->p_getfrom)) {
        rp->p_int_blocked = TRUE;
        return;
  }

  /* Destination is waiting for an interrupt.
   * Send it a message with source HARDWARE and type HARD_INT.
   * No more information can be reliably provided since interrupt messages
   * are not queued.
   */
  rp->p_messbuf->m_source = HARDWARE;
  rp->p_messbuf->m_type = HARD_INT;
  rp->p_flags &= ~RECEIVING;
  rp->p_int_blocked = FALSE;

   /* Make rp ready and run it unless a task is already running.  This is
    * ready(rp) in-line for speed.
    */
  if (rdy_head[TASK_Q] != NIL_PROC)
        rdy_tail[TASK_Q]->p_nextready = rp;
  else
        proc_ptr = rdy_head[TASK_Q] = rp;
  rdy_tail[TASK_Q] = rp;
  rp->p_nextready = NIL_PROC;
}

/*===========================================================================*
 *                              sys_call                                     *
 *===========================================================================*/
PUBLIC int sys_call(function, src_dest, m_ptr)
int function;                   /* SEND, RECEIVE, or BOTH */
int src_dest;                   /* source to receive from or dest to send to */
message *m_ptr;                 /* pointer to message */
{
/* The only system calls that exist in MINIX are sending and receiving
 * messages.  These are done by trapping to the kernel with an INT instruction.
```

```
 * The trap is caught and sys_call() is called to send or receive a message
 * (or both). The caller is always given by proc_ptr.
 */

  register struct proc *rp;
  int n;

  /* Check for bad system call parameters. */
  if (!isoksrc_dest(src_dest)) return(E_BAD_SRC);
  rp = proc_ptr;

  if (isuserp(rp) && function != BOTH) return(E_NO_PERM);

  /* The parameters are ok. Do the call. */
  if (function & SEND) {
        /* Function = SEND or BOTH. */
        n = mini_send(rp, src_dest, m_ptr);
        if (function == SEND || n != OK)
                return(n);      /* done, or SEND failed */
  }

  /* Function = RECEIVE or BOTH.
   * We have checked user calls are BOTH, and trust 'function' otherwise.
   */
  return(mini_rec(rp, src_dest, m_ptr));
}

/*===========================================================================*
 *                              mini_send                                    *
 *===========================================================================*/
PRIVATE int mini_send(caller_ptr, dest, m_ptr)
register struct proc *caller_ptr;       /* who is trying to send a message? */
int dest;                       /* to whom is message being sent? */
message *m_ptr;                 /* pointer to message buffer */
{
/* Send a message from 'caller_ptr' to 'dest'. If 'dest' is blocked waiting
 * for this message, copy the message to it and unblock 'dest'. If 'dest' is
 * not waiting at all, or is waiting for another source, queue 'caller_ptr'.
 */

  register struct proc *dest_ptr, *next_ptr;
  vir_bytes vb;                 /* message buffer pointer as vir_bytes */
  vir_clicks vlo, vhi;          /* virtual clicks containing message to send */

  /* User processes are only allowed to send to FS and MM.  Check for this. */
  if (isuserp(caller_ptr) && !issysentn(dest)) return(E_BAD_DEST);
  dest_ptr = proc_addr(dest);   /* pointer to destination's proc entry */
  if (dest_ptr->p_flags & P_SLOT_FREE) return(E_BAD_DEST);      /* dead dest */

  /* This check allows a message to be anywhere in data or stack or gap.
   * It will have to be made more elaborate later for machines which
   * don't have the gap mapped.
   */
  vb = (vir_bytes) m_ptr;
  vlo = vb >> CLICK_SHIFT;      /* vir click for bottom of message */
  vhi = (vb + MESS_SIZE - 1) >> CLICK_SHIFT;    /* vir click for top of msg */
  if (vlo < caller_ptr->p_map[D].mem_vir || vlo > vhi ||
      vhi >= caller_ptr->p_map[S].mem_vir + caller_ptr->p_map[S].mem_len)
        return(EFAULT);
```

```
  /* Check for deadlock by 'caller_ptr' and 'dest' sending to each other. */
  if (dest_ptr->p_flags & SENDING) {
        next_ptr = proc_addr(dest_ptr->p_sendto);
        while (TRUE) {
                if (next_ptr == caller_ptr) return(ELOCKED);
                if (next_ptr->p_flags & SENDING)
                        next_ptr = proc_addr(next_ptr->p_sendto);
                else
                        break;
        }
  }

  /* Check to see if 'dest' is blocked waiting for this message. */
  if ( (dest_ptr->p_flags & (RECEIVING | SENDING)) == RECEIVING &&
       (dest_ptr->p_getfrom == ANY ||
        dest_ptr->p_getfrom == proc_number(caller_ptr))) {
        /* Destination is indeed waiting for this message. */
        CopyMess(proc_number(caller_ptr), caller_ptr, m_ptr, dest_ptr,
                 dest_ptr->p_messbuf);
        dest_ptr->p_flags &= ~RECEIVING;        /* deblock destination */
        if (dest_ptr->p_flags == 0) ready(dest_ptr);
  } else {
        /* Destination is not waiting.  Block and queue caller. */
        caller_ptr->p_messbuf = m_ptr;
        if (caller_ptr->p_flags == 0) unready(caller_ptr);
        caller_ptr->p_flags |= SENDING;
        caller_ptr->p_sendto= dest;

        /* Process is now blocked.  Put in on the destination's queue. */
        if ( (next_ptr = dest_ptr->p_callerq) == NIL_PROC)
                dest_ptr->p_callerq = caller_ptr;
        else {
                while (next_ptr->p_sendlink != NIL_PROC)
                        next_ptr = next_ptr->p_sendlink;
                next_ptr->p_sendlink = caller_ptr;
        }
        caller_ptr->p_sendlink = NIL_PROC;
  }
  return(OK);
}

/*===========================================================================*
 *                              mini_rec                                     *
 *===========================================================================*/
PRIVATE int mini_rec(caller_ptr, src, m_ptr)
register struct proc *caller_ptr;       /* process trying to get message */
int src;                        /* which message source is wanted (or ANY) */
message *m_ptr;                 /* pointer to message buffer */
{
/* A process or task wants to get a message.  If one is already queued,
 * acquire it and deblock the sender.  If no message from the desired source
 * is available, block the caller.  No need to check parameters for validity.
 * Users calls are always sendrec(), and mini_send() has checked already.
 * Calls from the tasks, MM, and FS are trusted.
 */

  register struct proc *sender_ptr;
  register struct proc *previous_ptr;

  /* Check to see if a message from desired source is already available. */
```

```
  if (!(caller_ptr->p_flags & SENDING)) {
        /* Check caller queue. */
    for (sender_ptr = caller_ptr->p_callerq; sender_ptr != NIL_PROC;
         previous_ptr = sender_ptr, sender_ptr = sender_ptr->p_sendlink) {
        if (src == ANY || src == proc_number(sender_ptr)) {
                /* An acceptable message has been found. */
                CopyMess(proc_number(sender_ptr), sender_ptr,
                         sender_ptr->p_messbuf, caller_ptr, m_ptr);
                if (sender_ptr == caller_ptr->p_callerq)
                        caller_ptr->p_callerq = sender_ptr->p_sendlink;
                else
                        previous_ptr->p_sendlink = sender_ptr->p_sendlink;
                if ((sender_ptr->p_flags &= ~SENDING) == 0)
                        ready(sender_ptr);      /* deblock sender */
                return(OK);
        }
    }

    /* Check for blocked interrupt. */
    if (caller_ptr->p_int_blocked && isrxhardware(src)) {
        m_ptr->m_source = HARDWARE;
        m_ptr->m_type = HARD_INT;
        caller_ptr->p_int_blocked = FALSE;
        return(OK);
    }
  }

  /* No suitable message is available.  Block the process trying to receive. */
  caller_ptr->p_getfrom = src;
  caller_ptr->p_messbuf = m_ptr;
  if (caller_ptr->p_flags == 0) unready(caller_ptr);
  caller_ptr->p_flags |= RECEIVING;

  /* If MM has just blocked and there are kernel signals pending, now is the
   * time to tell MM about them, since it will be able to accept the message.
   */
  if (sig_procs > 0 && proc_number(caller_ptr) == MM_PROC_NR && src == ANY)
        inform();
  return(OK);
}

/*===========================================================================*
 *                              pick_proc                                    *
 *===========================================================================*/
PRIVATE void pick_proc()
{
/* Decide who to run now.  A new process is selected by setting 'proc_ptr'.
 * When a fresh user (or idle) process is selected, record it in 'bill_ptr',
 * so the clock task can tell who to bill for system time.
 */

  register struct proc *rp;     /* process to run */

  if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
        proc_ptr = rp;
        return;
  }
  if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
        proc_ptr = rp;
        return;
```

```
  }
  if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
        proc_ptr = rp;
        bill_ptr = rp;
        return;
  }
  /* No one is ready.  Run the idle task.  The idle task might be made an
   * always-ready user task to avoid this special case.
   */
  bill_ptr = proc_ptr = proc_addr(IDLE);
}

/*===========================================================================*
 *                              ready                                        *
 *===========================================================================*/
PRIVATE void ready(rp)
register struct proc *rp;       /* this process is now runnable */
{
/* Add 'rp' to the end of one of the queues of runnable processes. Three
 * queues are maintained:
 *   TASK_Q   - (highest priority) for runnable tasks
 *   SERVER_Q - (middle priority) for MM and FS only
 *   USER_Q   - (lowest priority) for user processes
 */

  if (istaskp(rp)) {
        if (rdy_head[TASK_Q] != NIL_PROC)
                /* Add to tail of nonempty queue. */
                rdy_tail[TASK_Q]->p_nextready = rp;
        else {
                proc_ptr =              /* run fresh task next */
                rdy_head[TASK_Q] = rp;  /* add to empty queue */
        }
        rdy_tail[TASK_Q] = rp;
        rp->p_nextready = NIL_PROC;     /* new entry has no successor */
        return;
  }
  if (!isuserp(rp)) {           /* others are similar */
        if (rdy_head[SERVER_Q] != NIL_PROC)
                rdy_tail[SERVER_Q]->p_nextready = rp;
        else
                rdy_head[SERVER_Q] = rp;
        rdy_tail[SERVER_Q] = rp;
        rp->p_nextready = NIL_PROC;
        return;
  }
  if (rdy_head[USER_Q] == NIL_PROC)
        rdy_tail[USER_Q] = rp;
  rp->p_nextready = rdy_head[USER_Q];
  rdy_head[USER_Q] = rp;
/*
  if (rdy_head[USER_Q] != NIL_PROC)
        rdy_tail[USER_Q]->p_nextready = rp;
  else
        rdy_head[USER_Q] = rp;
  rdy_tail[USER_Q] = rp;
  rp->p_nextready = NIL_PROC;
*/
}
```

```
/*===========================================================================*
 *                              unready                                      *
 *===========================================================================*/
PRIVATE void unready(rp)
register struct proc *rp;       /* this process is no longer runnable */
{
/* A process has blocked. */

  register struct proc *xp;
  register struct proc **qtail;  /* TASK_Q, SERVER_Q, or USER_Q rdy_tail */

  if (istaskp(rp)) {
        /* task stack still ok? */
        if (*rp->p_stguard != STACK_GUARD)
                panic("stack overrun by task", proc_number(rp));

        if ( (xp = rdy_head[TASK_Q]) == NIL_PROC) return;
        if (xp == rp) {
                /* Remove head of queue */
                rdy_head[TASK_Q] = xp->p_nextready;
                if (rp == proc_ptr) pick_proc();
                return;
        }
        qtail = &rdy_tail[TASK_Q];
  }
  else if (!isuserp(rp)) {
        if ( (xp = rdy_head[SERVER_Q]) == NIL_PROC) return;
        if (xp == rp) {
                rdy_head[SERVER_Q] = xp->p_nextready;
                pick_proc();
                return;
        }
        qtail = &rdy_tail[SERVER_Q];
  } else
  {
        if ( (xp = rdy_head[USER_Q]) == NIL_PROC) return;
        if (xp == rp) {
                rdy_head[USER_Q] = xp->p_nextready;
                pick_proc();
                return;
        }
        qtail = &rdy_tail[USER_Q];
  }

  /* Search body of queue.  A process can be made unready even if it is
   * not running by being sent a signal that kills it.
   */
  while (xp->p_nextready != rp)
        if ( (xp = xp->p_nextready) == NIL_PROC) return;
  xp->p_nextready = xp->p_nextready->p_nextready;
  if (*qtail == rp) *qtail = xp;
}

/*===========================================================================*
 *                              sched                                        *
 *===========================================================================*/
PRIVATE void sched()
{
/* The current process has run too long.  If another low priority (user)
 * process is runnable, put the current process on the end of the user queue,
```

```
 * possibly promoting another user to head of the queue.
 */

  if (rdy_head[USER_Q] == NIL_PROC) return;

  /* One or more user processes queued. */
  rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
  rdy_tail[USER_Q] = rdy_head[USER_Q];
  rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
  rdy_tail[USER_Q]->p_nextready = NIL_PROC;
  pick_proc();
}

/*===========================================================================*
 *                              lock_mini_send                               *
 *===========================================================================*/
PUBLIC int lock_mini_send(caller_ptr, dest, m_ptr)
struct proc *caller_ptr;        /* who is trying to send a message? */
int dest;                       /* to whom is message being sent? */
message *m_ptr;                 /* pointer to message buffer */
{
/* Safe gateway to mini_send() for tasks. */

  int result;

  switching = TRUE;
  result = mini_send(caller_ptr, dest, m_ptr);
  switching = FALSE;
  return(result);
}

/*===========================================================================*
 *                              lock_pick_proc                               *
 *===========================================================================*/
PUBLIC void lock_pick_proc()
{
/* Safe gateway to pick_proc() for tasks. */

  switching = TRUE;
  pick_proc();
  switching = FALSE;
}

/*===========================================================================*
 *                              lock_ready                                   *
 *===========================================================================*/
PUBLIC void lock_ready(rp)
struct proc *rp;                /* this process is now runnable */
{
/* Safe gateway to ready() for tasks. */

  switching = TRUE;
  ready(rp);
  switching = FALSE;
}


/*===========================================================================*
 *                              lock_unready                                 *
 *===========================================================================*/
```

```
PUBLIC void lock_unready(rp)
struct proc *rp;                /* this process is no longer runnable */
{
/* Safe gateway to unready() for tasks. */

  switching = TRUE;
  unready(rp);
  switching = FALSE;
}

/*===========================================================================*
 *                              lock_sched                                   *
 *===========================================================================*/
PUBLIC void lock_sched()
{
/* Safe gateway to sched() for tasks. */

  switching = TRUE;
  sched();
  switching = FALSE;
}

/*===========================================================================*
 *                              unhold                                       *
 *===========================================================================*/
PUBLIC void unhold()
{
/* Flush any held-up interrupts.  k_reenter must be 0.  held_head must not
 * be NIL_PROC.  Interrupts must be disabled.  They will be enabled but will
 * be disabled when this returns.
 */

  register struct proc *rp;     /* current head of held queue */

  if (switching) return;
  rp = held_head;
  do {
        if ( (held_head = rp->p_nextheld) == NIL_PROC) held_tail = NIL_PROC;
        rp->p_int_held = FALSE;
        unlock();               /* reduce latency; held queue may change! */
        interrupt(proc_number(rp));
        lock();                 /* protect the held queue again */
  }
  while ( (rp = held_head) != NIL_PROC);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/exception.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains a simple exception handler.  Exceptions in user
 * processes are converted to signals.  Exceptions in the kernel, MM and
 * FS cause a panic.
 */

#include "kernel.h"
#include <signal.h>
#include "proc.h"

/*==========================================================================*
 *                              exception                                   *
 *==========================================================================*/
PUBLIC void exception(vec_nr)
unsigned vec_nr;
{
/* An exception or unexpected interrupt has occurred. */

  struct ex_s {
        char *msg;
        int signum;
        int minprocessor;
  };
  static struct ex_s ex_data[] = {
        "Divide error", SIGFPE, 86,
        "Debug exception", SIGTRAP, 86,
        "Nonmaskable interrupt", SIGBUS, 86,
        "Breakpoint", SIGEMT, 86,
        "Overflow", SIGFPE, 86,
        "Bounds check", SIGFPE, 186,
        "Invalid opcode", SIGILL, 186,
        "Coprocessor not available", SIGFPE, 186,
        "Double fault", SIGBUS, 286,
        "Copressor segment overrun", SIGSEGV, 286,
        "Invalid TSS", SIGSEGV, 286,
        "Segment not present", SIGSEGV, 286,
        "Stack exception", SIGSEGV, 286,        /* STACK_FAULT already used */
        "General protection", SIGSEGV, 286,
        "Page fault", SIGSEGV, 386,             /* not close */
        NIL_PTR, SIGILL, 0,                     /* probably software trap */
        "Coprocessor error", SIGFPE, 386,
  };
  register struct ex_s *ep;
  struct proc *saved_proc;

  saved_proc= proc_ptr; /* Save proc_ptr, because it may be changed by debug
                         * statements.
                         */

  ep = &ex_data[vec_nr];

  if (vec_nr == 2) {            /* spurious NMI on some machines */
        printf("got spurious NMI\n");
        return;
  }
```

```
  if (k_reenter == 0 && isuserp(saved_proc)) {
        unlock();               /* this is protected like sys_call() */
        cause_sig(proc_number(saved_proc), ep->signum);
        return;
  }

  /* This is not supposed to happen. */
  if (ep->msg == NIL_PTR || processor < ep->minprocessor)
        printf("\nIntel-reserved exception %d\n", vec_nr);
  else
        printf("\n%s\n", ep->msg);
  printf("process number %d, pc = 0x%04x:0x%08x\n",
        proc_number(saved_proc),
        (unsigned) saved_proc->p_reg.cs,
        (unsigned) saved_proc->p_reg.pc);
  panic("exception in system code", NO_NUM);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/i8259.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains routines for initializing the 8259 interrupt controller:
 *      get_irq_handler: address of handler for a given interrupt
 *      put_irq_handler: register an interrupt handler
 *      intr_init:      initialize the interrupt controller(s)
 */

#include "kernel.h"

#define ICW1_AT         0x11    /* edge triggered, cascade, need ICW4 */
#define ICW1_PC         0x13    /* edge triggered, no cascade, need ICW4 */
#define ICW1_PS         0x19    /* level triggered, cascade, need ICW4 */
#define ICW4_AT         0x01    /* not SFNM, not buffered, normal EOI, 8086 */
#define ICW4_PC         0x09    /* not SFNM, buffered, normal EOI, 8086 */

FORWARD _PROTOTYPE( int spurious_irq, (int irq) );

#define set_vec(nr, addr)       ((void)0) /* kluge for protected mode */

/*===========================================================================*
 *                              intr_init                                    *
 *===========================================================================*/
PUBLIC void intr_init(mine)
int mine;
{
/* Initialize the 8259s, finishing with all interrupts disabled.  This is
 * only done in protected mode, in real mode we don't touch the 8259s, but
 * use the BIOS locations instead.  The flag "mine" is set if the 8259s are
 * to be programmed for Minix, or to be reset to what the BIOS expects.
 */

  int i;

  lock();
  /* The AT and newer PS/2 have two interrupt controllers, one master,
   * one slaved at IRQ 2.  (We don't have to deal with the PC that
```

```
    * has just one controller, because it must run in real mode.)
    */
  out_byte(INT_CTL, ps_mca ? ICW1_PS : ICW1_AT);
  out_byte(INT_CTLMASK, mine ? IRQ0_VECTOR : BIOS_IRQ0_VEC);
                                                /* ICW2 for master */
  out_byte(INT_CTLMASK, (1 << CASCADE_IRQ));    /* ICW3 tells slaves */
  out_byte(INT_CTLMASK, ICW4_AT);
  out_byte(INT_CTLMASK, ~(1 << CASCADE_IRQ));   /* IRQ 0-7 mask */
  out_byte(INT2_CTL, ps_mca ? ICW1_PS : ICW1_AT);
  out_byte(INT2_CTLMASK, mine ? IRQ8_VECTOR : BIOS_IRQ8_VEC);
                                                /* ICW2 for slave */
  out_byte(INT2_CTLMASK, CASCADE_IRQ);          /* ICW3 is slave nr */
  out_byte(INT2_CTLMASK, ICW4_AT);
  out_byte(INT2_CTLMASK, ~0);                   /* IRQ 8-15 mask */

  /* Initialize the table of interrupt handlers. */
  for (i = 0; i < NR_IRQ_VECTORS; i++) irq_table[i] = spurious_irq;
}

/*==========================================================================*
 *                              spurious_irq                                *
 *==========================================================================*/
PRIVATE int spurious_irq(irq)
int irq;
{
/* Default interrupt handler.  It complains a lot. */

  if (irq < 0 || irq >= NR_IRQ_VECTORS)
        panic("invalid call to spurious_irq", irq);

  printf("spurious irq %d\n", irq);

  return 1;     /* Reenable interrupt */
}

/*==========================================================================*
 *                              put_irq_handler                             *
 *==========================================================================*/
PUBLIC void put_irq_handler(irq, handler)
int irq;
irq_handler_t handler;
{
/* Register an interrupt handler. */

  if (irq < 0 || irq >= NR_IRQ_VECTORS)
        panic("invalid call to put_irq_handler", irq);

  if (irq_table[irq] == handler)
        return;         /* extra initialization */

  if (irq_table[irq] != spurious_irq)
        panic("attempt to register second irq handler for irq", irq);

  disable_irq(irq);
  if (!protected_mode) set_vec(BIOS_VECTOR(irq), irq_vec[irq]);
  irq_table[irq]= handler;
  irq_use |= 1 << irq;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/protect.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```c
/* This file contains code for initialization of protected mode, to initialize
 * code and data segment descriptors, and to initialize global descriptors
 * for local descriptors in the process table.
 */

#include "kernel.h"
#include "proc.h"
#include "protect.h"

#define INT_GATE_TYPE   (INT_286_GATE | DESC_386_BIT)
#define TSS_TYPE        (AVL_286_TSS  | DESC_386_BIT)

struct desctableptr_s {
  char limit[sizeof(u16_t)];
  char base[sizeof(u32_t)];             /* really u24_t + pad for 286 */
};

struct gatedesc_s {
  u16_t offset_low;
  u16_t selector;
  u8_t pad;                     /* |000|XXXXX| ig & trpg, |XXXXXXXX| task g */
  u8_t p_dpl_type;              /* |P|DL|0|TYPE| */
  u16_t offset_high;
};

struct tss_s {
  reg_t backlink;
  reg_t sp0;                    /* stack pointer to use during interrupt */
  reg_t ss0;                    /*   "   segment  "  "    "        "     */
  reg_t sp1;
  reg_t ss1;
  reg_t sp2;
  reg_t ss2;
  reg_t cr3;
  reg_t ip;
  reg_t flags;
  reg_t ax;
  reg_t cx;
  reg_t dx;
  reg_t bx;
  reg_t sp;
  reg_t bp;
  reg_t si;
  reg_t di;
  reg_t es;
  reg_t cs;
  reg_t ss;
  reg_t ds;
  reg_t fs;
  reg_t gs;
  reg_t ldt;
  u16_t trap;
  u16_t iobase;
};

```

```
PUBLIC struct segdesc_s gdt[GDT_SIZE];
PRIVATE struct gatedesc_s idt[IDT_SIZE];        /* zero-init so none present */
PUBLIC struct tss_s tss;        /* zero init */

FORWARD _PROTOTYPE( void int_gate, (unsigned vec_nr, phys_bytes base,
                unsigned dpl_type) );
FORWARD _PROTOTYPE( void sdesc, (struct segdesc_s *segdp, phys_bytes base,
                phys_bytes size) );

/*===========================================================================*
 *                              prot_init                                    *
 *===========================================================================*/
PUBLIC void prot_init()
{
/* Set up tables for protected mode.
 * All GDT slots are allocated at compile time.
 */

  phys_bytes code_bytes;
  phys_bytes data_bytes;
  struct gate_table_s *gtp;
  struct desctableptr_s *dtp;
  unsigned ldt_selector;
  register struct proc *rp;

  static struct gate_table_s {
        _PROTOTYPE( void (*gate), (void) );
        unsigned char vec_nr;
        unsigned char privilege;
  }
  gate_table[] = {
        divide_error, DIVIDE_VECTOR, INTR_PRIVILEGE,
        single_step_exception, DEBUG_VECTOR, INTR_PRIVILEGE,
        nmi, NMI_VECTOR, INTR_PRIVILEGE,
        breakpoint_exception, BREAKPOINT_VECTOR, USER_PRIVILEGE,
        overflow, OVERFLOW_VECTOR, USER_PRIVILEGE,
        bounds_check, BOUNDS_VECTOR, INTR_PRIVILEGE,
        inval_opcode, INVAL_OP_VECTOR, INTR_PRIVILEGE,
        copr_not_available, COPROC_NOT_VECTOR, INTR_PRIVILEGE,
        double_fault, DOUBLE_FAULT_VECTOR, INTR_PRIVILEGE,
        copr_seg_overrun, COPROC_SEG_VECTOR, INTR_PRIVILEGE,
        inval_tss, INVAL_TSS_VECTOR, INTR_PRIVILEGE,
        segment_not_present, SEG_NOT_VECTOR, INTR_PRIVILEGE,
        stack_exception, STACK_FAULT_VECTOR, INTR_PRIVILEGE,
        general_protection, PROTECTION_VECTOR, INTR_PRIVILEGE,
        page_fault, PAGE_FAULT_VECTOR, INTR_PRIVILEGE,
        copr_error, COPROC_ERR_VECTOR, INTR_PRIVILEGE,
        { hwint00, VECTOR( 0), INTR_PRIVILEGE },
        { hwint01, VECTOR( 1), INTR_PRIVILEGE },
        { hwint02, VECTOR( 2), INTR_PRIVILEGE },
        { hwint03, VECTOR( 3), INTR_PRIVILEGE },
        { hwint04, VECTOR( 4), INTR_PRIVILEGE },
        { hwint05, VECTOR( 5), INTR_PRIVILEGE },
        { hwint06, VECTOR( 6), INTR_PRIVILEGE },
        { hwint07, VECTOR( 7), INTR_PRIVILEGE },
        { hwint08, VECTOR( 8), INTR_PRIVILEGE },
        { hwint09, VECTOR( 9), INTR_PRIVILEGE },
        { hwint10, VECTOR(10), INTR_PRIVILEGE },
        { hwint11, VECTOR(11), INTR_PRIVILEGE },
        { hwint12, VECTOR(12), INTR_PRIVILEGE },
```

```
                { hwint13, VECTOR(13), INTR_PRIVILEGE },
                { hwint14, VECTOR(14), INTR_PRIVILEGE },
                { hwint15, VECTOR(15), INTR_PRIVILEGE },
        };

        /* This is called early and can't use tables set up by main(). */
        data_bytes = (phys_bytes) sizes[1] << CLICK_SHIFT;
        if (sizes[0] == 0)
                code_bytes = data_bytes;        /* common I&D */
        else
                code_bytes = (phys_bytes) sizes[0] << CLICK_SHIFT;

        /* Build gdt and idt pointers in GDT where the BIOS expects them. */
        dtp= (struct desctableptr_s *) &gdt[GDT_INDEX];
        * (u16_t *) dtp->limit = (sizeof gdt) - 1;
        * (u32_t *) dtp->base = vir2phys(gdt);

        dtp= (struct desctableptr_s *) &gdt[IDT_INDEX];
        * (u16_t *) dtp->limit = (sizeof idt) - 1;
        * (u32_t *) dtp->base = vir2phys(idt);

        /* Build segment descriptors for tasks and interrupt handlers. */
        init_codeseg(&gdt[CS_INDEX], code_base, code_bytes, INTR_PRIVILEGE);
        init_dataseg(&gdt[DS_INDEX], data_base, data_bytes, INTR_PRIVILEGE);
        init_dataseg(&gdt[ES_INDEX], 0L, 0L, TASK_PRIVILEGE);

        /* Build scratch descriptors for functions in klib88. */
        init_dataseg(&gdt[DS_286_INDEX], (phys_bytes) 0,
                     (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);
        init_dataseg(&gdt[ES_286_INDEX], (phys_bytes) 0,
                     (phys_bytes) MAX_286_SEG_SIZE, TASK_PRIVILEGE);

        /* Build local descriptors in GDT for LDT's in process table.
         * The LDT's are allocated at compile time in the process table, and
         * initialized whenever a process' map is initialized or changed.
         */
        for (rp = BEG_PROC_ADDR, ldt_selector = FIRST_LDT_INDEX * DESC_SIZE;
             rp < END_PROC_ADDR; ++rp, ldt_selector += DESC_SIZE) {
                init_dataseg(&gdt[ldt_selector / DESC_SIZE], vir2phys(rp->p_ldt),
                             (phys_bytes) sizeof rp->p_ldt, INTR_PRIVILEGE);
                gdt[ldt_selector / DESC_SIZE].access = PRESENT | LDT;
                rp->p_ldt_sel = ldt_selector;
        }

        /* Build main TSS.
         * This is used only to record the stack pointer to be used after an
         * interrupt.
         * The pointer is set up so that an interrupt automatically saves the
         * current process's registers ip:cs:f:sp:ss in the correct slots in the
         * process table.
         */
        tss.ss0 = DS_SELECTOR;
        init_dataseg(&gdt[TSS_INDEX], vir2phys(&tss), (phys_bytes) sizeof tss,
                                                        INTR_PRIVILEGE);
        gdt[TSS_INDEX].access = PRESENT | (INTR_PRIVILEGE << DPL_SHIFT) | TSS_TYPE;
        tss.iobase = sizeof tss;        /* empty i/o permissions map */

        /* Build descriptors for interrupt gates in IDT. */
        for (gtp = &gate_table[0];
             gtp < &gate_table[sizeof gate_table / sizeof gate_table[0]]; ++gtp) {
```

```
        int_gate(gtp->vec_nr, (phys_bytes) (vir_bytes) gtp->gate,
                 PRESENT | INT_GATE_TYPE | (gtp->privilege << DPL_SHIFT));
  }
  int_gate(SYS_VECTOR, (phys_bytes) (vir_bytes) p_s_call,
           PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
  int_gate(LEVEL0_VECTOR, (phys_bytes) (vir_bytes) level0_call,
           PRESENT | (TASK_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
  int_gate(SYS386_VECTOR, (phys_bytes) (vir_bytes) s_call,
           PRESENT | (USER_PRIVILEGE << DPL_SHIFT) | INT_GATE_TYPE);
}

/*===========================================================================*
 *                              init_codeseg                                 *
 *===========================================================================*/
PUBLIC void init_codeseg(segdp, base, size, privilege)
register struct segdesc_s *segdp;
phys_bytes base;
phys_bytes size;
int privilege;
{
/* Build descriptor for a code segment. */

  sdesc(segdp, base, size);
  segdp->access = (privilege << DPL_SHIFT)
                | (PRESENT | SEGMENT | EXECUTABLE | READABLE);
                /* CONFORMING = 0, ACCESSED = 0 */
}

/*===========================================================================*
 *                              init_dataseg                                 *
 *===========================================================================*/
PUBLIC void init_dataseg(segdp, base, size, privilege)
register struct segdesc_s *segdp;
phys_bytes base;
phys_bytes size;
int privilege;
{
/* Build descriptor for a data segment. */

  sdesc(segdp, base, size);
  segdp->access = (privilege << DPL_SHIFT) | (PRESENT | SEGMENT | WRITEABLE);
                /* EXECUTABLE = 0, EXPAND_DOWN = 0, ACCESSED = 0 */
}

/*===========================================================================*
 *                              sdesc                                        *
 *===========================================================================*/
PRIVATE void sdesc(segdp, base, size)
register struct segdesc_s *segdp;
phys_bytes base;
phys_bytes size;
{
/* Fill in the size fields (base, limit and granularity) of a descriptor. */

  segdp->base_low = base;
  segdp->base_middle = base >> BASE_MIDDLE_SHIFT;
  segdp->base_high = base >> BASE_HIGH_SHIFT;
  --size;                       /* convert to a limit, 0 size means 4G */
  if (size > BYTE_GRAN_MAX) {
        segdp->limit_low = size >> PAGE_GRAN_SHIFT;
```

```
        segdp->granularity = GRANULAR | (size >>
                                     (PAGE_GRAN_SHIFT + GRANULARITY_SHIFT));
  } else {
        segdp->limit_low = size;
        segdp->granularity = size >> GRANULARITY_SHIFT;
  }
  segdp->granularity |= DEFAULT;        /* means BIG for data seg */
}

/*===========================================================================*
 *                              seg2phys                                     *
 *===========================================================================*/
PUBLIC phys_bytes seg2phys(seg)
U16_t seg;
{
/* Return the base address of a segment, with seg being either a 8086 segment
 * register, or a 286/386 segment selector.
 */
  phys_bytes base;
  struct segdesc_s *segdp;

  if (!protected_mode) {
        base = hclick_to_physb(seg);
  } else {
        segdp = &gdt[seg >> 3];
        base = segdp->base_low | ((u32_t) segdp->base_middle << 16);
        base |= ((u32_t) segdp->base_high << 24);
  }
  return base;
}

/*===========================================================================*
 *                              int_gate                                     *
 *===========================================================================*/
PRIVATE void int_gate(vec_nr, base, dpl_type)
unsigned vec_nr;
phys_bytes base;
unsigned dpl_type;
{
/* Build descriptor for an interrupt gate. */

  register struct gatedesc_s *idp;

  idp = &idt[vec_nr];
  idp->offset_low = base;
  idp->selector = CS_SELECTOR;
  idp->p_dpl_type = dpl_type;
  idp->offset_high = base >> OFFSET_HIGH_SHIFT;
}

/*===========================================================================*
 *                              enable_iop                                   *
 *===========================================================================*/
PUBLIC void enable_iop(pp)
struct proc *pp;
{
/* Allow a user process to use I/O instructions.  Change the I/O Permission
 * Level bits in the psw. These specify least-privileged Current Permission
 * Level allowed to execute I/O instructions. Users and servers have CPL 3.
 * You can't have less privilege than that. Kernel has CPL 0, tasks CPL 1.
```

```
   */
    pp->p_reg.psw |= 0x3000;
  }
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/klib.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#
! Chooses between the 8086 and 386 versions of the low level kernel code.

#include <minix/config.h>
#if _WORD_SIZE == 2
#include "klib88.s"
#else
#include "klib386.s"
#endif
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/klib386.s
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
#
! sections

.sect .text; .sect .rom; .sect .data; .sect .bss

#include <minix/config.h>
#include <minix/const.h>
#include "const.h"
#include "sconst.h"
#include "protect.h"

! This file contains a number of assembly code utility routines needed by the
! kernel.  They are:

.define _monitor        ! exit Minix and return to the monitor
.define _check_mem      ! check a block of memory, return the valid size
.define _cp_mess        ! copies messages from source to destination
.define _exit           ! dummy for library routines
.define __exit          ! dummy for library routines
.define ___exit         ! dummy for library routines
.define ___main         ! dummy for GCC
.define _in_byte        ! read a byte from a port and return it
.define _in_word        ! read a word from a port and return it
.define _out_byte       ! write a byte to a port
.define _out_word       ! write a word to a port
.define _port_read      ! transfer data from (disk controller) port to memory
.define _port_read_byte ! likewise byte by byte
.define _port_write     ! transfer data from memory to (disk controller) port
.define _port_write_byte ! likewise byte by byte
.define _lock           ! disable interrupts
.define _unlock         ! enable interrupts
.define _enable_irq     ! enable an irq at the 8259 controller
.define _disable_irq    ! disable an irq
.define _phys_copy      ! copy data from anywhere to anywhere in memory
.define _mem_rdw        ! copy one word from [segment:offset]
```

```
.define _reset          ! reset the system
.define _mem_vid_copy   ! copy data to video ram
.define _vid_vid_copy   ! move data in video ram
.define _level0         ! call a function at level 0

! The routines only guarantee to preserve the registers the C compiler
! expects to be preserved (ebx, esi, edi, ebp, esp, segment registers, and
! direction bit in the flags).

! imported variables

.sect .bss
.extern _mon_return, _mon_sp
.extern _irq_use
.extern _blank_color
.extern _ext_memsize
.extern _gdt
.extern _low_memsize
.extern _sizes
.extern _vid_seg
.extern _vid_size
.extern _vid_mask
.extern _level0_func

.sect .text
!*===========================================================================*
!*                              monitor                                      *
!*===========================================================================*
! PUBLIC void monitor();
! Return to the monitor.

_monitor:
        mov     eax, (_reboot_code)     ! address of new parameters
        mov     esp, (_mon_sp)          ! restore monitor stack pointer
    o16 mov     dx, SS_SELECTOR         ! monitor data segment
        mov     ds, dx
        mov     es, dx
        mov     fs, dx
        mov     gs, dx
        mov     ss, dx
        pop     edi
        pop     esi
        pop     ebp
    o16 retf                            ! return to the monitor


!*===========================================================================*
!*                              check_mem                                    *
!*===========================================================================*
! PUBLIC phys_bytes check_mem(phys_bytes base, phys_bytes size);
! Check a block of memory, return the amount valid.
! Only every 16th byte is checked.
! An initial size of 0 means everything.
! This really should do some alias checks.

CM_DENSITY      =       16
CM_LOG_DENSITY  =       4
TEST1PATTERN    =       0x55            ! memory test pattern 1
TEST2PATTERN    =       0xAA            ! memory test pattern 2

```

```
CHKM_ARGS       =       4 + 4 + 4       ! 4 + 4
!                       ds ebx eip      base size

_check_mem:
        push    ebx
        push    ds
    o16 mov     ax, FLAT_DS_SELECTOR
        mov     ds, ax
        mov     eax, CHKM_ARGS(esp)
        mov     ebx, eax
        mov     ecx, CHKM_ARGS+4(esp)
        shr     ecx, CM_LOG_DENSITY
cm_loop:
        movb    dl, TEST1PATTERN
        xchgb   dl, (eax)               ! write test pattern, remember original
        xchgb   dl, (eax)               ! restore original, read test pattern
        cmpb    dl, TEST1PATTERN        ! must agree if good real memory
        jnz     cm_exit                 ! if different, memory is unusable
        movb    dl, TEST2PATTERN
        xchgb   dl, (eax)
        xchgb   dl, (eax)
        add     eax, CM_DENSITY
        cmpb    dl, TEST2PATTERN
        loopz   cm_loop
cm_exit:
        sub     eax, ebx
        pop     ds
        pop     ebx
        ret


!*===========================================================================*
!*                              cp_mess                                      *
!*===========================================================================*
! PUBLIC void cp_mess(int src, phys_clicks src_clicks, vir_bytes src_offset,
!                     phys_clicks dst_clicks, vir_bytes dst_offset);
! This routine makes a fast copy of a message from anywhere in the address
! space to anywhere else.  It also copies the source address provided as a
! parameter to the call into the first word of the destination message.
!
! Note that the message size, "Msize" is in DWORDS (not bytes) and must be set
! correctly.  Changing the definition of message in the type file and not
! changing it here will lead to total disaster.

CM_ARGS =       4 + 4 + 4 + 4 + 4       ! 4 + 4 + 4 + 4 + 4
!               es  ds edi esi eip      proc scl sof dcl dof

        .align  16
_cp_mess:
        cld
        push    esi
        push    edi
        push    ds
        push    es

        mov     eax, FLAT_DS_SELECTOR
        mov     ds, ax
        mov     es, ax

        mov     esi, CM_ARGS+4(esp)             ! src clicks
```

```
        shl     esi, CLICK_SHIFT
        add     esi, CM_ARGS+4+4(esp)           ! src offset
        mov     edi, CM_ARGS+4+4+4(esp)         ! dst clicks
        shl     edi, CLICK_SHIFT
        add     edi, CM_ARGS+4+4+4+4(esp)       ! dst offset

        mov     eax, CM_ARGS(esp)       ! process number of sender
        stos                            ! copy number of sender to dest message
        add     esi, 4                  ! do not copy first word
        mov     ecx, Msize - 1          ! remember, first word does not count
        rep
        movs                            ! copy the message

        pop     es
        pop     ds
        pop     edi
        pop     esi
        ret                             ! that is all folks!


!*===========================================================================*
!*                              exit                                         *
!*===========================================================================*
! PUBLIC void exit();
! Some library routines use exit, so provide a dummy version.
! Actual calls to exit cannot occur in the kernel.
! GNU CC likes to call ___main from main() for nonobvious reasons.

_exit:
__exit:
___exit:
        sti
        jmp     ___exit

___main:
        ret


!*===========================================================================*
!*                              in_byte                                      *
!*===========================================================================*
! PUBLIC unsigned in_byte(port_t port);
! Read an (unsigned) byte from the i/o port  port  and return it.

        .align  16
_in_byte:
        mov     edx, 4(esp)             ! port
        sub     eax, eax
        inb     dx                      ! read 1 byte
        ret


!*===========================================================================*
!*                              in_word                                      *
!*===========================================================================*
! PUBLIC unsigned in_word(port_t port);
! Read an (unsigned) word from the i/o port  port  and return it.

        .align  16
_in_word:
```

```
        mov     edx, 4(esp)             ! port
        sub     eax, eax
    o16 in      dx                      ! read 1 word
        ret


!*===========================================================================*
!*                              out_byte                                     *
!*===========================================================================*
! PUBLIC void out_byte(port_t port, u8_t value);
! Write  value  (cast to a byte)  to the I/O port  port.

        .align  16
_out_byte:
        mov     edx, 4(esp)             ! port
        movb    al, 4+4(esp)            ! value
        outb    dx                      ! output 1 byte
        ret


!*===========================================================================*
!*                              out_word                                     *
!*===========================================================================*
! PUBLIC void out_word(Port_t port, U16_t value);
! Write  value  (cast to a word)  to the I/O port  port.

        .align  16
_out_word:
        mov     edx, 4(esp)             ! port
        mov     eax, 4+4(esp)           ! value
    o16 out     dx                      ! output 1 word
        ret


!*===========================================================================*
!*                              port_read                                    *
!*===========================================================================*
! PUBLIC void port_read(port_t port, phys_bytes destination, unsigned bytcount);
! Transfer data from (hard disk controller) port to memory.

PR_ARGS =       4 + 4 + 4               ! 4 + 4 + 4
!               es edi eip              port dst len

        .align  16
_port_read:
        cld
        push    edi
        push    es
        mov     ecx, FLAT_DS_SELECTOR
        mov     es, cx
        mov     edx, PR_ARGS(esp)       ! port to read from
        mov     edi, PR_ARGS+4(esp)     ! destination addr
        mov     ecx, PR_ARGS+4+4(esp)   ! byte count
        shr     ecx, 1                  ! word count
        rep                             ! (hardware cannot handle dwords)
    o16 ins                             ! read everything
        pop     es
        pop     edi
        ret

```

```

!*===========================================================================*
!*                              port_read_byte                               *
!*===========================================================================*
! PUBLIC void port_read_byte(port_t port, phys_bytes destination,
!                                                 unsigned bytcount);
! Transfer data from port to memory.

PR_ARGS_B =     4 + 4 + 4               ! 4 + 4 + 4
!               es edi eip              port dst len

_port_read_byte:
        cld
        push    edi
        push    es
        mov     ecx, FLAT_DS_SELECTOR
        mov     es, cx
        mov     edx, PR_ARGS_B(esp)
        mov     edi, PR_ARGS_B+4(esp)
        mov     ecx, PR_ARGS_B+4+4(esp)
        rep
        insb
        pop     es
        pop     edi
        ret


!*===========================================================================*
!*                              port_write                                   *
!*===========================================================================*
! PUBLIC void port_write(port_t port, phys_bytes source, unsigned bytcount);
! Transfer data from memory to (hard disk controller) port.

PW_ARGS =       4 + 4 + 4               ! 4 + 4 + 4
!               es edi eip              port src len

        .align  16
_port_write:
        cld
        push    esi
        push    ds
        mov     ecx, FLAT_DS_SELECTOR
        mov     ds, cx
        mov     edx, PW_ARGS(esp)       ! port to write to
        mov     esi, PW_ARGS+4(esp)     ! source addr
        mov     ecx, PW_ARGS+4+4(esp)   ! byte count
        shr     ecx, 1                  ! word count
        rep                             ! (hardware cannot handle dwords)
    o16 outs                            ! write everything
        pop     ds
        pop     esi
        ret


!*===========================================================================*
!*                              port_write_byte                              *
!*===========================================================================*
! PUBLIC void port_write_byte(port_t port, phys_bytes source,
!                                                 unsigned bytcount);
! Transfer data from memory to port.
```

```

PW_ARGS_B =     4 + 4 + 4               ! 4 + 4 + 4
!               es edi eip              port src len

_port_write_byte:
        cld
        push    esi
        push    ds
        mov     ecx, FLAT_DS_SELECTOR
        mov     ds, cx
        mov     edx, PW_ARGS_B(esp)
        mov     esi, PW_ARGS_B+4(esp)
        mov     ecx, PW_ARGS_B+4+4(esp)
        rep
        outsb
        pop     ds
        pop     esi
        ret


!*===========================================================================*
!*                              lock                                         *
!*===========================================================================*
! PUBLIC void lock();
! Disable CPU interrupts.

        .align  16
_lock:
        cli                             ! disable interrupts
        ret


!*===========================================================================*
!*                              unlock                                       *
!*===========================================================================*
! PUBLIC void unlock();
! Enable CPU interrupts.

        .align  16
_unlock:
        sti
        ret


!*==========================================================================*
!*                              enable_irq                                  *
!*==========================================================================*/
! PUBLIC void enable_irq(unsigned irq)
! Enable an interrupt request line by clearing an 8259 bit.
! Equivalent code for irq < 8:
!       out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) & ~(1 << irq));

        .align  16
_enable_irq:
        mov     ecx, 4(esp)             ! irq
        pushf
        cli
        movb    ah, ~1
        rolb    ah, cl                  ! ah = ~(1 << (irq % 8))
        cmpb    cl, 8
```

```
        jae     enable_8                ! enable irq >= 8 at the slave 8259
enable_0:
        inb     INT_CTLMASK
        andb    al, ah
        outb    INT_CTLMASK             ! clear bit at master 8259
        popf
        ret
        .align  4
enable_8:
        inb     INT2_CTLMASK
        andb    al, ah
        outb    INT2_CTLMASK            ! clear bit at slave 8259
        popf
        ret


!*==========================================================================*
!*                              disable_irq                                 *
!*==========================================================================*/
! PUBLIC int disable_irq(unsigned irq)
! Disable an interrupt request line by setting an 8259 bit.
! Equivalent code for irq < 8:
!       out_byte(INT_CTLMASK, in_byte(INT_CTLMASK) | (1 << irq));
! Returns true iff the interrupt was not already disabled.

        .align  16
_disable_irq:
        mov     ecx, 4(esp)             ! irq
        pushf
        cli
        movb    ah, 1
        rolb    ah, cl                  ! ah = (1 << (irq % 8))
        cmpb    cl, 8
        jae     disable_8               ! disable irq >= 8 at the slave 8259
disable_0:
        inb     INT_CTLMASK
        testb   al, ah
        jnz     dis_already             ! already disabled?
        orb     al, ah
        outb    INT_CTLMASK             ! set bit at master 8259
        popf
        mov     eax, 1                  ! disabled by this function
        ret
disable_8:
        inb     INT2_CTLMASK
        testb   al, ah
        jnz     dis_already             ! already disabled?
        orb     al, ah
        outb    INT2_CTLMASK            ! set bit at slave 8259
        popf
        mov     eax, 1                  ! disabled by this function
        ret
dis_already:
        popf
        xor     eax, eax                ! already disabled
        ret


!*===========================================================================*
!*                              phys_copy                                    *
```

```
!*===========================================================================*
! PUBLIC void phys_copy(phys_bytes source, phys_bytes destination,
!                       phys_bytes bytecount);
! Copy a block of physical memory.

PC_ARGS =       4 + 4 + 4 + 4   ! 4 + 4 + 4
!               es edi esi eip   src dst len

        .align  16
_phys_copy:
        cld
        push    esi
        push    edi
        push    es

        mov     eax, FLAT_DS_SELECTOR
        mov     es, ax

        mov     esi, PC_ARGS(esp)
        mov     edi, PC_ARGS+4(esp)
        mov     eax, PC_ARGS+4+4(esp)

        cmp     eax, 10                 ! avoid align overhead for small counts
        jb      pc_small
        mov     ecx, esi                ! align source, hope target is too
        neg     ecx
        and     ecx, 3                  ! count for alignment
        sub     eax, ecx
        rep
   eseg movsb
        mov     ecx, eax
        shr     ecx, 2                  ! count of dwords
        rep
   eseg movs
        and     eax, 3
pc_small:
        xchg    ecx, eax                ! remainder
        rep
   eseg movsb

        pop     es
        pop     edi
        pop     esi
        ret


!*===========================================================================*
!*                              mem_rdw                                      *
!*===========================================================================*
! PUBLIC u16_t mem_rdw(U16_t segment, u16_t *offset);
! Load and return word at far pointer segment:offset.

        .align  16
_mem_rdw:
        mov     cx, ds
        mov     ds, 4(esp)              ! segment
        mov     eax, 4+4(esp)           ! offset
        movzx   eax, (eax)              ! word to return
        mov     ds, cx
        ret
```

```


!*===========================================================================*
!*                              reset                                        *
!*===========================================================================*
! PUBLIC void reset();
! Reset the system by loading IDT with offset 0 and interrupting.

_reset:
        lidt    (idt_zero)
        int     3               ! anything goes, the 386 will not like it
.sect .data
idt_zero:       .data4  0, 0
.sect .text


!*===========================================================================*
!*                              mem_vid_copy                                 *
!*===========================================================================*
! PUBLIC void mem_vid_copy(u16 *src, unsigned dst, unsigned count);
!
! Copy count characters from kernel memory to video memory.  Src, dst and
! count are character (word) based video offsets and counts.  If src is null
! then screen memory is blanked by filling it with blank_color.

MVC_ARGS        =       4 + 4 + 4 + 4   ! 4 + 4 + 4
!                       es edi esi eip   src dst ct

_mem_vid_copy:
        push    esi
        push    edi
        push    es
        mov     esi, MVC_ARGS(esp)      ! source
        mov     edi, MVC_ARGS+4(esp)    ! destination
        mov     edx, MVC_ARGS+4+4(esp)  ! count
        mov     es, (_vid_seg)          ! destination is video segment
        cld                             ! make sure direction is up
mvc_loop:
        and     edi, (_vid_mask)        ! wrap address
        mov     ecx, edx                ! one chunk to copy
        mov     eax, (_vid_size)
        sub     eax, edi
        cmp     ecx, eax
        jbe     0f
        mov     ecx, eax                ! ecx = min(ecx, vid_size - edi)
0:      sub     edx, ecx                ! count -= ecx
        shl     edi, 1                  ! byte address
        test    esi, esi                ! source == 0 means blank the screen
        jz      mvc_blank
mvc_copy:
        rep                             ! copy words to video memory
    o16 movs
        jmp     mvc_test
mvc_blank:
        mov     eax, (_blank_color)     ! ax = blanking character
        rep
    o16 stos                            ! copy blanks to video memory
        !jmp    mvc_test
mvc_test:
        shr     edi, 1                  ! word addresses
```

```
        test    edx, edx
        jnz     mvc_loop
mvc_done:
        pop     es
        pop     edi
        pop     esi
        ret


!*===========================================================================*
!*                              vid_vid_copy                                 *
!*===========================================================================*
! PUBLIC void vid_vid_copy(unsigned src, unsigned dst, unsigned count);
!
! Copy count characters from video memory to video memory.  Handle overlap.
! Used for scrolling, line or character insertion and deletion.  Src, dst
! and count are character (word) based video offsets and counts.

VVC_ARGS        =       4 + 4 + 4 + 4   ! 4 + 4 + 4
!                       es edi esi eip   src dst ct

_vid_vid_copy:
        push    esi
        push    edi
        push    es
        mov     esi, VVC_ARGS(esp)      ! source
        mov     edi, VVC_ARGS+4(esp)    ! destination
        mov     edx, VVC_ARGS+4+4(esp)  ! count
        mov     es, (_vid_seg)          ! use video segment
        cmp     esi, edi                ! copy up or down?
        jb      vvc_down
vvc_up:
        cld                             ! direction is up
vvc_uploop:
        and     esi, (_vid_mask)        ! wrap addresses
        and     edi, (_vid_mask)
        mov     ecx, edx                ! one chunk to copy
        mov     eax, (_vid_size)
        sub     eax, esi
        cmp     ecx, eax
        jbe     0f
        mov     ecx, eax                ! ecx = min(ecx, vid_size - esi)
0:      mov     eax, (_vid_size)
        sub     eax, edi
        cmp     ecx, eax
        jbe     0f
        mov     ecx, eax                ! ecx = min(ecx, vid_size - edi)
0:      sub     edx, ecx                ! count -= ecx
        shl     esi, 1
        shl     edi, 1                  ! byte addresses
        rep
eseg o16 movs                           ! copy video words
        shr     esi, 1
        shr     edi, 1                  ! word addresses
        test    edx, edx
        jnz     vvc_uploop              ! again?
        jmp     vvc_done
vvc_down:
        std                             ! direction is down
        lea     esi, -1(esi)(edx*1)     ! start copying at the top
```

```
        lea     edi, -1(edi)(edx*1)
vvc_downloop:
        and     esi, (_vid_mask)        ! wrap addresses
        and     edi, (_vid_mask)
        mov     ecx, edx                ! one chunk to copy
        lea     eax, 1(esi)
        cmp     ecx, eax
        jbe     0f
        mov     ecx, eax                ! ecx = min(ecx, esi + 1)
0:      lea     eax, 1(edi)
        cmp     ecx, eax
        jbe     0f
        mov     ecx, eax                ! ecx = min(ecx, edi + 1)
0:      sub     edx, ecx                ! count -= ecx
        shl     esi, 1
        shl     edi, 1                  ! byte addresses
        rep
eseg o16 movs                           ! copy video words
        shr     esi, 1
        shr     edi, 1                  ! word addresses
        test    edx, edx
        jnz     vvc_downloop            ! again?
        cld                             ! C compiler expect up
        !jmp    vvc_done
vvc_done:
        pop     es
        pop     edi
        pop     esi
        ret


!*===========================================================================*
!*                              level0                                       *
!*===========================================================================*
! PUBLIC void level0(void (*func)(void))
! Call a function at permission level 0.  This allows kernel tasks to do
! things that are only possible at the most privileged CPU level.
!
_level0:
        mov     eax, 4(esp)
        mov     (_level0_func), eax
        int     LEVEL0_VECTOR
        ret
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/misc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains a collection of miscellaneous procedures:
 *      mem_init:       initialize memory tables.  Some memory is reported
 *                      by the BIOS, some is guesstimated and checked later
 *      env_parse       parse environment variable.
 *      bad_assertion   for debugging
 *      bad_compare     for debugging
 */

#include "kernel.h"
#include "assert.h"
```

```
#include <stdlib.h>
#include <minix/com.h>

#define EM_BASE     0x100000L   /* base of extended memory on AT's */
#define SHADOW_BASE 0xFA0000L   /* base of RAM shadowing ROM on some AT's */
#define SHADOW_MAX  0x060000L   /* maximum usable shadow memory (16M limit) */

/*==========================================================================*
 *                              mem_init                                    *
 *==========================================================================*/
PUBLIC void mem_init()
{
/* Initialize the memory size tables.  This is complicated by fragmentation
 * and different access strategies for protected mode.  There must be a
 * chunk at 0 big enough to hold Minix proper.  For 286 and 386 processors,
 * there can be extended memory (memory above 1MB).  This usually starts at
 * 1MB, but there may be another chunk just below 16MB, reserved under DOS
 * for shadowing ROM, but available to Minix if the hardware can be re-mapped.
 * In protected mode, extended memory is accessible assuming CLICK_SIZE is
 * large enough, and is treated as ordinary memory.
 */

  u32_t ext_clicks;
  phys_clicks max_clicks;

  /* Get the size of ordinary memory from the BIOS. */
  mem[0].size = k_to_click(low_memsize);        /* base = 0 */

  if (pc_at && protected_mode) {
        /* Get the size of extended memory from the BIOS.  This is special
         * except in protected mode, but protected mode is now normal.
         * Note that no more than 16M can be addressed in 286 mode, so make
         * sure that the highest memory address fits in a short when counted
         * in clicks.
         */
        ext_clicks = k_to_click((u32_t) ext_memsize);
        max_clicks = USHRT_MAX - (EM_BASE >> CLICK_SHIFT);
        mem[1].size = MIN(ext_clicks, max_clicks);
        mem[1].base = EM_BASE >> CLICK_SHIFT;

        if (ext_memsize <= (unsigned) ((SHADOW_BASE - EM_BASE) / 1024)
                        && check_mem(SHADOW_BASE, SHADOW_MAX) == SHADOW_MAX) {
                /* Shadow ROM memory. */
                mem[2].size = SHADOW_MAX >> CLICK_SHIFT;
                mem[2].base = SHADOW_BASE >> CLICK_SHIFT;
        }
  }

  /* Total system memory. */
  tot_mem_size = mem[0].size + mem[1].size + mem[2].size;
}

/*==========================================================================*
 *                              env_parse                                   *
 *==========================================================================*/
PUBLIC int env_parse(env, fmt, field, param, min, max)
char *env;              /* environment variable to inspect */
char *fmt;              /* template to parse it with */
int field;              /* field number of value to return */
long *param;            /* address of parameter to get */
```

```
long min, max;          /* minimum and maximum values for the parameter */
{
/* Parse an environment variable setting, something like "DPETH0=300:3".
 * Panic if the parsing fails.  Return EP_UNSET if the environment variable
 * is not set, EP_OFF if it is set to "off", EP_ON if set to "on" or a
 * field is left blank, or EP_SET if a field is given (return value through
 * *param).  Commas and colons may be used in the environment and format
 * string, fields in the environment string may be empty, and punctuation
 * may be missing to skip fields.  The format string contains characters
 * 'd', 'o', 'x' and 'c' to indicate that 10, 8, 16, or 0 is used as the
 * last argument to strtol.
 */

  char *val, *end;
  long newpar;
  int i = 0, radix, r;

  if ((val = k_getenv(env)) == NIL_PTR) return(EP_UNSET);
  if (strcmp(val, "off") == 0) return(EP_OFF);
  if (strcmp(val, "on") == 0) return(EP_ON);

  r = EP_ON;
  for (;;) {
        while (*val == ' ') val++;

        if (*val == 0) return(r);       /* the proper exit point */

        if (*fmt == 0) break;           /* too many values */

        if (*val == ',' || *val == ':') {
                /* Time to go to the next field. */
                if (*fmt == ',' || *fmt == ':') i++;
                if (*fmt++ == *val) val++;
        } else {
                /* Environment contains a value, get it. */
                switch (*fmt) {
                case 'd':       radix =   10;   break;
                case 'o':       radix =  010;   break;
                case 'x':       radix = 0x10;   break;
                case 'c':       radix =    0;   break;
                default:        goto badenv;
                }
                newpar = strtol(val, &end, radix);

                if (end == val) break;  /* not a number */
                val = end;

                if (i == field) {
                        /* The field requested. */
                        if (newpar < min || newpar > max) break;
                        *param = newpar;
                        r = EP_SET;
                }
        }
  }
badenv:
  printf("Bad environment setting: '%s = %s'\n", env, k_getenv(env));
  panic("", NO_NUM);
  /*NOTREACHED*/
}
```

```
#if DEBUG
/*===========================================================================*
 *                              bad_assertion                               *
 *===========================================================================*/
PUBLIC void bad_assertion(file, line, what)
char *file;
int line;
char *what;
{
  printf("panic at %s(%d): assertion \"%s\" failed\n", file, line, what);
  panic(NULL, NO_NUM);
}

/*===========================================================================*
 *                              bad_compare                                 *
 *===========================================================================*/
PUBLIC void bad_compare(file, line, lhs, what, rhs)
char *file;
int line;
int lhs;
char *what;
int rhs;
{
  printf("panic at %s(%d): compare (%d) %s (%d) failed\n",
        file, line, lhs, what, rhs);
  panic(NULL, NO_NUM);
}
#endif /* DEBUG */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                              src/kernel/driver.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Types and constants shared between the generic and device dependent
 * device driver code.
 */

#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"
#include <minix/partition.h>

/* Info about and entry points into the device dependent code. */
struct driver {
  _PROTOTYPE( char *(*dr_name), (void) );
  _PROTOTYPE( int (*dr_open), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_close), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( int (*dr_ioctl), (struct driver *dp, message *m_ptr) );
  _PROTOTYPE( struct device *(*dr_prepare), (int device) );
  _PROTOTYPE( int (*dr_schedule), (int proc_nr, struct iorequest_s *request) );
  _PROTOTYPE( int (*dr_finish), (void) );
  _PROTOTYPE( void (*dr_cleanup), (void) );
  _PROTOTYPE( void (*dr_geometry), (struct partition *entry) );
};

#if (CHIP == INTEL)

/* Number of bytes you can DMA before hitting a 64K boundary: */
```

```c
#define dma_bytes_left(phys)    \
   ((unsigned) (sizeof(int) == 2 ? 0 : 0x10000) - (unsigned) ((phys) & 0xFFFF))

#endif /* CHIP == INTEL */

/* Base and size of a partition in bytes. */
struct device {
  unsigned long dv_base;
  unsigned long dv_size;
};

#define NIL_DEV         ((struct device *) 0)

/* Functions defined by driver.c: */
_PROTOTYPE( void driver_task, (struct driver *dr) );
_PROTOTYPE( int do_rdwt, (struct driver *dr, message *m_ptr) );
_PROTOTYPE( int do_vrdwt, (struct driver *dr, message *m_ptr) );
_PROTOTYPE( char *no_name, (void) );
_PROTOTYPE( int do_nop, (struct driver *dp, message *m_ptr) );
_PROTOTYPE( int nop_finish, (void) );
_PROTOTYPE( void nop_cleanup, (void) );
_PROTOTYPE( void clock_mess, (int ticks, watchdog_t func) );
_PROTOTYPE( int do_diocntl, (struct driver *dr, message *m_ptr) );

/* Parameters for the disk drive. */
#define SECTOR_SIZE      512    /* physical sector size in bytes */
#define SECTOR_SHIFT       9    /* for division */
#define SECTOR_MASK      511    /* and remainder */

/* Size of the DMA buffer buffer in bytes. */
#define DMA_BUF_SIZE    (DMA_SECTORS * SECTOR_SIZE)

#if (CHIP == INTEL)
extern u8_t *tmp_buf;                   /* the DMA buffer */
#else
extern u8_t tmp_buf[];                  /* the DMA buffer */
#endif
extern phys_bytes tmp_phys;             /* phys address of DMA buffer */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/driver.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```c
/* This file contains device independent device driver interface.
 *                                                      Author: Kees J. Bot.
 *
 * The drivers support the following operations (using message format m2):
 *
 *    m_type      DEVICE    PROC_NR     COUNT    POSITION  ADRRESS
 * ----------------------------------------------------------------
 * |  DEV_OPEN  | device  | proc nr |         |         |         |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_CLOSE | device  | proc nr |         |         |         |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_READ  | device  | proc nr |  bytes  |  offset | buf ptr |
 * |------------+---------+---------+---------+---------+---------|
 * |  DEV_WRITE | device  | proc nr |  bytes  |  offset | buf ptr |
 * |------------+---------+---------+---------+---------+---------|
```

```
 * |SCATTERED_IO| device  | proc nr | requests|         | iov ptr |
 * ----------------------------------------------------------------
 * |  DEV_IOCTL | device  | proc nr |func code|         | buf ptr |
 * ----------------------------------------------------------------
 *
 * The file contains one entry point:
 *
 *   driver_task:       called by the device dependent task entry
 *
 *
 * Constructed 92/04/02 by Kees J. Bot from the old AT wini and floppy driver.
 */

#include "kernel.h"
#include <sys/ioctl.h>
#include "driver.h"

#define BUF_EXTRA       0

/* Claim space for variables. */
PRIVATE u8_t buffer[(unsigned) 2 * DMA_BUF_SIZE + BUF_EXTRA];
u8_t *tmp_buf;                  /* the DMA buffer eventually */
phys_bytes tmp_phys;            /* phys address of DMA buffer */

FORWARD _PROTOTYPE( void init_buffer, (void) );

/*===========================================================================*
 *                              driver_task                                  *
 *===========================================================================*/
PUBLIC void driver_task(dp)
struct driver *dp;      /* Device dependent entry points. */
{
/* Main program of any device driver task. */

  int r, caller, proc_nr;
  message mess;

  init_buffer();        /* Get a DMA buffer. */

  /* Here is the main loop of the disk task.  It waits for a message, carries
   * it out, and sends a reply.
   */

  while (TRUE) {
        /* First wait for a request to read or write a disk block. */
        receive(ANY, &mess);

        caller = mess.m_source;
        proc_nr = mess.PROC_NR;

        switch (caller) {
        case HARDWARE:
                /* Leftover interrupt. */
                continue;
        case FS_PROC_NR:
                /* The only legitimate caller. */
                break;
        default:
                printf("%s: got message from %d\n", (*dp->dr_name)(), caller);
                continue;
```

```
        }

        /* Now carry out the work. */
        switch(mess.m_type) {
            case DEV_OPEN:      r = (*dp->dr_open)(dp, &mess);  break;
            case DEV_CLOSE:     r = (*dp->dr_close)(dp, &mess); break;
            case DEV_IOCTL:     r = (*dp->dr_ioctl)(dp, &mess); break;

            case DEV_READ:
            case DEV_WRITE:     r = do_rdwt(dp, &mess);         break;

            case SCATTERED_IO:  r = do_vrdwt(dp, &mess);        break;
            default:            r = EINVAL;                     break;
        }

        /* Clean up leftover state. */
        (*dp->dr_cleanup)();

        /* Finally, prepare and send the reply message. */
        mess.m_type = TASK_REPLY;
        mess.REP_PROC_NR = proc_nr;

        mess.REP_STATUS = r;    /* # of bytes transferred or error code */
        send(caller, &mess);    /* send reply to caller */
  }
}

/*===========================================================================*
 *                              init_buffer                                  *
 *===========================================================================*/
PRIVATE void init_buffer()
{
/* Select a buffer that can safely be used for dma transfers.  It may also
 * be used to read partition tables and such.  Its absolute address is
 * 'tmp_phys', the normal address is 'tmp_buf'.
 */

  tmp_buf = buffer;
  tmp_phys = vir2phys(buffer);

  if (tmp_phys == 0) panic("no DMA buffer", NO_NUM);

  if (dma_bytes_left(tmp_phys) < DMA_BUF_SIZE) {
        /* First half of buffer crosses a 64K boundary, can't DMA into that */
        tmp_buf += DMA_BUF_SIZE;
        tmp_phys += DMA_BUF_SIZE;
  }
}

/*===========================================================================*
 *                              do_rdwt                                      *
 *===========================================================================*/
PUBLIC int do_rdwt(dp, m_ptr)
struct driver *dp;              /* device dependent entry points */
message *m_ptr;                 /* pointer to read or write message */
{
/* Carry out a single read or write request. */
  struct iorequest_s ioreq;
  int r;

```

```
  if (m_ptr->COUNT <= 0) return(EINVAL);

  if ((*dp->dr_prepare)(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);

  ioreq.io_request = m_ptr->m_type;
  ioreq.io_buf = m_ptr->ADDRESS;
  ioreq.io_position = m_ptr->POSITION;
  ioreq.io_nbytes = m_ptr->COUNT;

  r = (*dp->dr_schedule)(m_ptr->PROC_NR, &ioreq);

  if (r == OK) (void) (*dp->dr_finish)();

  r = ioreq.io_nbytes;
  return(r < 0 ? r : m_ptr->COUNT - r);
}

/*===========================================================================*
 *                              do_vrdwt                                     *
 *===========================================================================*/
PUBLIC int do_vrdwt(dp, m_ptr)
struct driver *dp;      /* device dependent entry points */
message *m_ptr;         /* pointer to read or write message */
{
/* Fetch a vector of i/o requests.  Handle requests one at a time.  Return
 * status in the vector.
 */

  struct iorequest_s *iop;
  static struct iorequest_s iovec[NR_IOREQS];
  phys_bytes iovec_phys;
  unsigned nr_requests;
  int request;
  int r;
  phys_bytes user_iovec_phys;

  nr_requests = m_ptr->COUNT;

  if (nr_requests > sizeof iovec / sizeof iovec[0])
        panic("FS passed too big an I/O vector", nr_requests);

  iovec_phys = vir2phys(iovec);
  user_iovec_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
                        (vir_bytes) (nr_requests * sizeof iovec[0]));

  if (user_iovec_phys == 0)
        panic("FS passed a bad I/O vector", (int) m_ptr->ADDRESS);

  phys_copy(user_iovec_phys, iovec_phys,
                           (phys_bytes) nr_requests * sizeof iovec[0]);

  if ((*dp->dr_prepare)(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);

  for (request = 0, iop = iovec; request < nr_requests; request++, iop++) {
        if ((r = (*dp->dr_schedule)(m_ptr->PROC_NR, iop)) != OK) break;
  }

  if (r == OK) (void) (*dp->dr_finish)();

  phys_copy(iovec_phys, user_iovec_phys,
```

```
                              (phys_bytes) nr_requests * sizeof iovec[0]);
  return(OK);
}

/*===========================================================================*
 *                              no_name                                      *
 *===========================================================================*/
PUBLIC char *no_name()
{
/* If no specific name for the device. */

  return(tasktab[proc_number(proc_ptr) + NR_TASKS].name);
}

/*============================================================================*
 *                              do_nop                                        *
 *============================================================================*/
PUBLIC int do_nop(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Nothing there, or nothing to do. */

  switch (m_ptr->m_type) {
  case DEV_OPEN:        return(ENODEV);
  case DEV_CLOSE:       return(OK);
  case DEV_IOCTL:       return(ENOTTY);
  default:              return(EIO);
  }
}

/*===========================================================================*
 *                              nop_finish                                   *
 *===========================================================================*/
PUBLIC int nop_finish()
{
/* Nothing to finish, all the work has been done by dp->dr_schedule. */
  return(OK);
}

/*===========================================================================*
 *                              nop_cleanup                                  *
 *===========================================================================*/
PUBLIC void nop_cleanup()
{
/* Nothing to clean up. */
}

/*===========================================================================*
 *                              clock_mess                                   *
 *===========================================================================*/
PUBLIC void clock_mess(ticks, func)
int ticks;                      /* how many clock ticks to wait */
watchdog_t func;                /* function to call upon time out */
{
/* Send the clock task a message. */

  message mess;

  mess.m_type = SET_ALARM;
```

```
  mess.CLOCK_PROC_NR = proc_number(proc_ptr);
  mess.DELTA_TICKS = (long) ticks;
  mess.FUNC_TO_CALL = (sighandler_t) func;
  sendrec(CLOCK, &mess);
}

/*===========================================================================*
 *                              do_diocntl                                   *
 *===========================================================================*/
PUBLIC int do_diocntl(dp, m_ptr)
struct driver *dp;
message *m_ptr;                 /* pointer to ioctl request */
{
/* Carry out a partition setting/getting request. */
  struct device *dv;
  phys_bytes user_phys, entry_phys;
  struct partition entry;

  if (m_ptr->REQUEST != DIOCSETP && m_ptr->REQUEST != DIOCGETP) return(ENOTTY);

  /* Decode the message parameters. */
  if ((dv = (*dp->dr_prepare)(m_ptr->DEVICE)) == NIL_DEV) return(ENXIO);

  user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, sizeof(entry));
  if (user_phys == 0) return(EFAULT);

  entry_phys = vir2phys(&entry);

  if (m_ptr->REQUEST == DIOCSETP) {
        /* Copy just this one partition table entry. */
        phys_copy(user_phys, entry_phys, (phys_bytes) sizeof(entry));
        dv->dv_base = entry.base;
        dv->dv_size = entry.size;
  } else {
        /* Return a partition table entry and the geometry of the drive. */
        entry.base = dv->dv_base;
        entry.size = dv->dv_size;
        (*dp->dr_geometry)(&entry);
        phys_copy(entry_phys, user_phys, (phys_bytes) sizeof(entry));
  }
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/drvlib.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* IBM device driver definitions                        Author: Kees J. Bot
 *                                                              7 Dec 1995
 */

#include <ibm/partition.h>

_PROTOTYPE( void partition, (struct driver *dr, int device, int style) );

/* BIOS parameter table layout. */
#define bp_cylinders(t)         (* (u16_t *) (&(t)[0]))
```

```
#define bp_heads(t)             (* (u8_t *)  (&(t)[2]))
#define bp_reduced_wr(t)        (* (u16_t *) (&(t)[3]))
#define bp_precomp(t)           (* (u16_t *) (&(t)[5]))
#define bp_max_ecc(t)           (* (u8_t *)  (&(t)[7]))
#define bp_ctlbyte(t)           (* (u8_t *)  (&(t)[8]))
#define bp_landingzone(t)       (* (u16_t *) (&(t)[12]))
#define bp_sectors(t)           (* (u8_t *)  (&(t)[14]))

/* Miscellaneous. */
#define DEV_PER_DRIVE   (1 + NR_PARTITIONS)
#define MINOR_hd1a      128
#define MINOR_fd0a      (28<<2)
#define P_FLOPPY        0
#define P_PRIMARY       1
#define P_SUB           2
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/drvlib.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* IBM device driver utility functions.                 Author: Kees J. Bot
 *                                                                7 Dec 1995
 * Entry point:
 *   partition: partition a disk to the partition table(s) on it.
 */

#include "kernel.h"
#include "driver.h"
#include "drvlib.h"


FORWARD _PROTOTYPE( void extpartition, (struct driver *dp, int extdev,
                                                unsigned long extbase) );
FORWARD _PROTOTYPE( int get_part_table, (struct driver *dp, int device,
                        unsigned long offset, struct part_entry *table) );
FORWARD _PROTOTYPE( void sort, (struct part_entry *table) );


/*===========================================================================*
 *                              partition                                    *
 *===========================================================================*/
PUBLIC void partition(dp, device, style)
struct driver *dp;      /* device dependent entry points */
int device;             /* device to partition */
int style;              /* partitioning style: floppy, primary, sub. */
{
/* This routine is called on first open to initialize the partition tables
 * of a device.  It makes sure that each partition falls safely within the
 * device's limits.  Depending on the partition style we are either making
 * floppy partitions, primary partitions or subpartitions.  Only primary
 * partitions are sorted, because they are shared with other operating
 * systems that expect this.
 */
  struct part_entry table[NR_PARTITIONS], *pe;
  int disk, par;
```

```
  struct device *dv;
  unsigned long base, limit, part_limit;

  /* Get the geometry of the device to partition */
  if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV || dv->dv_size == 0) return;
  base = dv->dv_base >> SECTOR_SHIFT;
  limit = base + (dv->dv_size >> SECTOR_SHIFT);

  /* Read the partition table for the device. */
  if (!get_part_table(dp, device, 0L, table)) return;

  /* Compute the device number of the first partition. */
  switch (style) {
  case P_FLOPPY:
        device += MINOR_fd0a;
        break;
  case P_PRIMARY:
        sort(table);            /* sort a primary partition table */
        device += 1;
        break;
  case P_SUB:
        disk = device / DEV_PER_DRIVE;
        par = device % DEV_PER_DRIVE - 1;
        device = MINOR_hd1a + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;
  }

  /* Find an array of devices. */
  if ((dv = (*dp->dr_prepare)(device)) == NIL_DEV) return;

  /* Set the geometry of the partitions from the partition table. */
  for (par = 0; par < NR_PARTITIONS; par++, dv++) {
        /* Shrink the partition to fit within the device. */
        pe = &table[par];
        part_limit = pe->lowsec + pe->size;
        if (part_limit < pe->lowsec) part_limit = limit;
        if (part_limit > limit) part_limit = limit;
        if (pe->lowsec < base) pe->lowsec = base;
        if (part_limit < pe->lowsec) part_limit = pe->lowsec;

        dv->dv_base = pe->lowsec << SECTOR_SHIFT;
        dv->dv_size = (part_limit - pe->lowsec) << SECTOR_SHIFT;

        if (style == P_PRIMARY) {
                /* Each Minix primary partition can be subpartitioned. */
                if (pe->sysind == MINIX_PART)
                        partition(dp, device + par, P_SUB);

                /* An extended partition has logical partitions. */
                if (pe->sysind == EXT_PART)
                        extpartition(dp, device + par, pe->lowsec);
        }
  }
}


/*===========================================================================*
 *                              extpartition                                 *
 *===========================================================================*/
PRIVATE void extpartition(dp, extdev, extbase)
struct driver *dp;      /* device dependent entry points */
```

```
int extdev;             /* extended partition to scan */
unsigned long extbase;  /* sector offset of the base extended partition */
{
/* Extended partitions cannot be ignored alas, because people like to move
 * files to and from DOS partitions.  Avoid reading this code, it's no fun.
 */
  struct part_entry table[NR_PARTITIONS], *pe;
  int subdev, disk, par;
  struct device *dv;
  unsigned long offset, nextoffset;

  disk = extdev / DEV_PER_DRIVE;
  par = extdev % DEV_PER_DRIVE - 1;
  subdev = MINOR_hd1a + (disk * NR_PARTITIONS + par) * NR_PARTITIONS;

  offset = 0;
  do {
        if (!get_part_table(dp, extdev, offset, table)) return;
        sort(table);

        /* The table should contain one logical partition and optionally
         * another extended partition.  (It's a linked list.)
         */
        nextoffset = 0;
        for (par = 0; par < NR_PARTITIONS; par++) {
                pe = &table[par];
                if (pe->sysind == EXT_PART) {
                        nextoffset = pe->lowsec;
                } else
                if (pe->sysind != NO_PART) {
                        if ((dv = (*dp->dr_prepare)(subdev)) == NIL_DEV) return;

                        dv->dv_base = (extbase + offset
                                        + pe->lowsec) << SECTOR_SHIFT;
                        dv->dv_size = pe->size << SECTOR_SHIFT;

                        /* Out of devices? */
                        if (++subdev % NR_PARTITIONS == 0) return;
                }
        }
  } while ((offset = nextoffset) != 0);
}


/*===========================================================================*
 *                              get_part_table                              *
 *===========================================================================*/
PRIVATE int get_part_table(dp, device, offset, table)
struct driver *dp;
int device;
unsigned long offset;           /* sector offset to the table */
struct part_entry *table;       /* four entries */
{
/* Read the partition table for the device, return true iff there were no
 * errors.
 */
  message mess;

  mess.DEVICE = device;
  mess.POSITION = offset << SECTOR_SHIFT;
```

```
  mess.COUNT = SECTOR_SIZE;
  mess.ADDRESS = (char *) tmp_buf;
  mess.PROC_NR = proc_number(proc_ptr);
  mess.m_type = DEV_READ;

  if (do_rdwt(dp, &mess) != SECTOR_SIZE) {
        printf("%s: can't read partition table\n", (*dp->dr_name)());
        return 0;
  }
  if (tmp_buf[510] != 0x55 || tmp_buf[511] != 0xAA) {
        /* Invalid partition table. */
        return 0;
  }
  memcpy(table, (tmp_buf + PART_TABLE_OFF), NR_PARTITIONS * sizeof(table[0]));
  return 1;
}


/*===========================================================================*
 *                              sort                                         *
 *===========================================================================*/
PRIVATE void sort(table)
struct part_entry *table;
{
/* Sort a partition table. */
  struct part_entry *pe, tmp;
  int n = NR_PARTITIONS;

  do {
        for (pe = table; pe < table + NR_PARTITIONS-1; pe++) {
                if (pe[0].sysind == NO_PART
                        || (pe[0].lowsec > pe[1].lowsec
                                        && pe[1].sysind != NO_PART)) {
                        tmp = pe[0]; pe[0] = pe[1]; pe[1] = tmp;
                }
        }
  } while (--n > 0);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/memory.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the device dependent part of the drivers for the
 * following special files:
 *     /dev/null        - null device (data sink)
 *     /dev/mem         - absolute memory
 *     /dev/kmem        - kernel virtual memory
 *     /dev/ram         - RAM disk
 *
 * The file contains one entry point:
 *
 *   mem_task:  main entry when system is brought up
 *
 *  Changes:
 *      20 Apr  1992 by Kees J. Bot: device dependent/independent split
 */

```

```
#include "kernel.h"
#include "driver.h"
#include <sys/ioctl.h>

#define NR_RAMS            4    /* number of RAM-type devices */

PRIVATE struct device m_geom[NR_RAMS];  /* Base and size of each RAM disk */
PRIVATE int m_device;           /* current device */

FORWARD _PROTOTYPE( struct device *m_prepare, (int device) );
FORWARD _PROTOTYPE( int m_schedule, (int proc_nr, struct iorequest_s *iop) );
FORWARD _PROTOTYPE( int m_do_open, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( void m_init, (void) );
FORWARD _PROTOTYPE( int m_ioctl, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( void m_geometry, (struct partition *entry) );


/* Entry points to this driver. */
PRIVATE struct driver m_dtab = {
  no_name,      /* current device's name */
  m_do_open,    /* open or mount */
  do_nop,       /* nothing on a close */
  m_ioctl,      /* specify ram disk geometry */
  m_prepare,    /* prepare for I/O on a given minor device */
  m_schedule,   /* do the I/O */
  nop_finish,   /* schedule does the work, no need to be smart */
  nop_cleanup,  /* nothing's dirty */
  m_geometry,   /* memory device "geometry" */
};


/*===========================================================================*
 *                              mem_task                                     *
 *===========================================================================*/
PUBLIC void mem_task()
{
  m_init();
  driver_task(&m_dtab);
}


/*===========================================================================*
 *                              m_prepare                                    *
 *===========================================================================*/
PRIVATE struct device *m_prepare(device)
int device;
{
/* Prepare for I/O on a device. */

  if (device < 0 || device >= NR_RAMS) return(NIL_DEV);
  m_device = device;

  return(&m_geom[device]);
}


/*===========================================================================*
 *                              m_schedule                                   *
 *===========================================================================*/
PRIVATE int m_schedule(proc_nr, iop)
```

```
int proc_nr;                    /* process doing the request */
struct iorequest_s *iop;        /* pointer to read or write request */
{
/* Read or write /dev/null, /dev/mem, /dev/kmem, or /dev/ram. */

  int device, count, opcode;
  phys_bytes mem_phys, user_phys;
  struct device *dv;

  /* Type of request */
  opcode = iop->io_request & ~OPTIONAL_IO;

  /* Get minor device number and check for /dev/null. */
  device = m_device;
  dv = &m_geom[device];

  /* Determine address where data is to go or to come from. */
  user_phys = numap(proc_nr, (vir_bytes) iop->io_buf,
                                                (vir_bytes) iop->io_nbytes);
  if (user_phys == 0) return(iop->io_nbytes = EINVAL);

  if (device == NULL_DEV) {
        /* /dev/null: Black hole. */
        if (opcode == DEV_WRITE) iop->io_nbytes = 0;
        count = 0;
  } else {
        /* /dev/mem, /dev/kmem, or /dev/ram: Check for EOF */
        if (iop->io_position >= dv->dv_size) return(OK);
        count = iop->io_nbytes;
        if (iop->io_position + count > dv->dv_size)
                count = dv->dv_size - iop->io_position;
  }

  /* Set up 'mem_phys' for /dev/mem, /dev/kmem, or /dev/ram */
  mem_phys = dv->dv_base + iop->io_position;

  /* Book the number of bytes to be transferred in advance. */
  iop->io_nbytes -= count;

  if (count == 0) return(OK);

  /* Copy the data. */
  if (opcode == DEV_READ)
        phys_copy(mem_phys, user_phys, (phys_bytes) count);
  else
        phys_copy(user_phys, mem_phys, (phys_bytes) count);

  return(OK);
}


/*===========================================================================*
 *                              m_do_open                                    *
 *===========================================================================*/
PRIVATE int m_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Check device number on open.  Give I/O privileges to a process opening
 * /dev/mem or /dev/kmem.
```

```
 */

  if (m_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);

  if (m_device == MEM_DEV || m_device == KMEM_DEV)
        enable_iop(proc_addr(m_ptr->PROC_NR));

  return(OK);
}


/*===========================================================================*
 *                              m_init                                       *
 *===========================================================================*/
PRIVATE void m_init()
{
  /* Initialize this task. */
  extern int _end;

  m_geom[KMEM_DEV].dv_base = vir2phys(0);
  m_geom[KMEM_DEV].dv_size = vir2phys(&_end);

#if (CHIP == INTEL)
  if (!protected_mode) {
        m_geom[MEM_DEV].dv_size =   0x100000;   /* 1M for 8086 systems */
  } else {
#if _WORD_SIZE == 2
        m_geom[MEM_DEV].dv_size =  0x1000000;   /* 16M for 286 systems */
#else
        m_geom[MEM_DEV].dv_size = 0xFFFFFFFF;   /* 4G-1 for 386 systems */
#endif
  }
#endif
}


/*===========================================================================*
 *                              m_ioctl                                      *
 *===========================================================================*/
PRIVATE int m_ioctl(dp, m_ptr)
struct driver *dp;
message *m_ptr;                 /* pointer to read or write message */
{
/* Set parameters for one of the RAM disks. */

  unsigned long bytesize;
  unsigned base, size;
  struct memory *memp;
  static struct psinfo psinfo = { NR_TASKS, NR_PROCS, (vir_bytes) proc, 0, 0 };
  phys_bytes psinfo_phys;

  switch (m_ptr->REQUEST) {
  case MIOCRAMSIZE:
        /* FS sets the RAM disk size. */
        if (m_ptr->PROC_NR != FS_PROC_NR) return(EPERM);

        bytesize = m_ptr->POSITION * BLOCK_SIZE;
        size = (bytesize + CLICK_SHIFT-1) >> CLICK_SHIFT;

        /* Find a memory chunk big enough for the RAM disk. */
```

```
		memp= &mem[NR_MEMS];
		while ((--memp)->size < size) {
			if (memp == mem) panic("RAM disk is too big", NO_NUM);
		}
		base = memp->base;
		memp->base += size;
		memp->size -= size;

		m_geom[RAM_DEV].dv_base = (unsigned long) base << CLICK_SHIFT;
		m_geom[RAM_DEV].dv_size = bytesize;
		break;
  case MIOCSPSINFO:
		/* MM or FS set the address of their process table. */
		if (m_ptr->PROC_NR == MM_PROC_NR) {
			psinfo.mproc = (vir_bytes) m_ptr->ADDRESS;
		} else
		if (m_ptr->PROC_NR == FS_PROC_NR) {
			psinfo.fproc = (vir_bytes) m_ptr->ADDRESS;
		} else {
			return(EPERM);
		}
		break;
  case MIOCGPSINFO:
		/* The ps program wants the process table addresses. */
		psinfo_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS,
							sizeof(psinfo));
		if (psinfo_phys == 0) return(EFAULT);
		phys_copy(vir2phys(&psinfo), psinfo_phys, (phys_bytes) sizeof(psinfo));
		break;
  default:
		return(do_diocntl(&m_dtab, m_ptr));
  }
  return(OK);
}


/*===========================================================================*
 *                              m_geometry                                   *
 *===========================================================================*/
PRIVATE void m_geometry(entry)
struct partition *entry;
{
  /* Memory devices don't have a geometry, but the outside world insists. */
  entry->cylinders = (m_geom[m_device].dv_size >> SECTOR_SHIFT) / (64 * 32);
  entry->heads = 64;
  entry->sectors = 32;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/wini.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/*      wini.c - choose a winchester driver.            Author: Kees J. Bot
 *                                                              28 May 1994
 * Several different winchester drivers may be compiled
 * into the kernel, but only one may run.  That one is chosen here using
 * the boot variable 'hd'.
 */

#include "kernel.h"
#include "driver.h"

#if ENABLE_WINI

/* Map driver name to task function. */
struct hdmap {
  char          *name;
  task_t        *task;
} hdmap[] = {

#if ENABLE_AT_WINI
  { "at",       at_winchester_task      },
#endif

#if ENABLE_BIOS_WINI
  { "bios",     bios_winchester_task    },
#endif

#if ENABLE_ESDI_WINI
  { "esdi",     esdi_winchester_task    },
#endif

#if ENABLE_XT_WINI
  { "xt",       xt_winchester_task      },
#endif

};


/*===========================================================================*
 *                              winchester_task                              *
 *===========================================================================*/
PUBLIC void winchester_task()
{
  /* Call the default or selected winchester task. */
  char *hd;
  struct hdmap *map;

  hd = k_getenv("hd");

  for (map = hdmap; map < hdmap + sizeof(hdmap)/sizeof(hdmap[0]); map++) {
        if (hd == NULL || strcmp(hd, map->name) == 0) {
                /* Run the selected winchester task. */
                (*map->task)();
        }
  }
  panic("no hd driver", NO_NUM);
```

```
}
#endif /* ENABLE_WINI */
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/at_wini.c
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the device dependent part of a driver for the IBM-AT
 * winchester controller.
 * It was written by Adri Koppes.
 *
 * The file contains one entry point:
 *
 *   at_winchester_task:        main entry when system is brought up
 *
 *
 * Changes:
 *      13 Apr 1992 by Kees J. Bot: device dependent/independent split.
 */

#include "kernel.h"
#include "driver.h"
#include "drvlib.h"

#if ENABLE_AT_WINI

/* I/O Ports used by winchester disk controllers. */

/* Read and write registers */
#define REG_BASE0       0x1F0   /* base register of controller 0 */
#define REG_BASE1       0x170   /* base register of controller 1 */
#define REG_DATA            0   /* data register (offset from the base reg.) */
#define REG_PRECOMP         1   /* start of write precompensation */
#define REG_COUNT           2   /* sectors to transfer */
#define REG_SECTOR          3   /* sector number */
#define REG_CYL_LO          4   /* low byte of cylinder number */
#define REG_CYL_HI          5   /* high byte of cylinder number */
#define REG_LDH             6   /* lba, drive and head */
#define   LDH_DEFAULT           0xA0    /* ECC enable, 512 bytes per sector */
#define   LDH_LBA               0x40    /* Use LBA addressing */
#define   ldh_init(drive)       (LDH_DEFAULT | ((drive) << 4))

/* Read only registers */
#define REG_STATUS          7   /* status */
#define   STATUS_BSY            0x80    /* controller busy */
#define   STATUS_RDY            0x40    /* drive ready */
#define   STATUS_WF             0x20    /* write fault */
#define   STATUS_SC             0x10    /* seek complete (obsolete) */
#define   STATUS_DRQ            0x08    /* data transfer request */
#define   STATUS_CRD            0x04    /* corrected data */
#define   STATUS_IDX            0x02    /* index pulse */
#define   STATUS_ERR            0x01    /* error */
#define REG_ERROR           1   /* error code */
#define   ERROR_BB              0x80    /* bad block */
#define   ERROR_ECC             0x40    /* bad ecc bytes */
#define   ERROR_ID              0x10    /* id not found */
#define   ERROR_AC              0x04    /* aborted command */
```

```
#define   ERROR_TK              0x02    /* track zero error */
#define   ERROR_DM              0x01    /* no data address mark */

/* Write only registers */
#define REG_COMMAND         7   /* command */
#define   CMD_IDLE              0x00    /* for w_command: drive idle */
#define   CMD_RECALIBRATE       0x10    /* recalibrate drive */
#define   CMD_READ              0x20    /* read data */
#define   CMD_WRITE             0x30    /* write data */
#define   CMD_READVERIFY        0x40    /* read verify */
#define   CMD_FORMAT            0x50    /* format track */
#define   CMD_SEEK              0x70    /* seek cylinder */
#define   CMD_DIAG              0x90    /* execute device diagnostics */
#define   CMD_SPECIFY           0x91    /* specify parameters */
#define   ATA_IDENTIFY          0xEC    /* identify drive */
#define REG_CTL         0x206   /* control register */
#define   CTL_NORETRY           0x80    /* disable access retry */
#define   CTL_NOECC             0x40    /* disable ecc retry */
#define   CTL_EIGHTHEADS        0x08    /* more than eight heads */
#define   CTL_RESET             0x04    /* reset controller */
#define   CTL_INTDISABLE        0x02    /* disable interrupts */

/* Interrupt request lines. */
#define AT_IRQ0         14      /* interrupt number for controller 0 */
#define AT_IRQ1         15      /* interrupt number for controller 1 */

/* Common command block */
struct command {
  u8_t  precomp;        /* REG_PRECOMP, etc. */
  u8_t  count;
  u8_t  sector;
  u8_t  cyl_lo;
  u8_t  cyl_hi;
  u8_t  ldh;
  u8_t  command;
};


/* Error codes */
#define ERR              (-1)   /* general error */
#define ERR_BAD_SECTOR   (-2)   /* block marked bad detected */

/* Some controllers don't interrupt, the clock will wake us up. */
#define WAKEUP          (32*HZ) /* drive may be out for 31 seconds max */

/* Miscellaneous. */
#define MAX_DRIVES         4    /* this driver supports 4 drives (hd0 - hd19) */
#if _WORD_SIZE > 2
#define MAX_SECS         256    /* controller can transfer this many sectors */
#else
#define MAX_SECS         127    /* but not to a 16 bit process */
#endif
#define MAX_ERRORS         4    /* how often to try rd/wt before quitting */
#define NR_DEVICES      (MAX_DRIVES * DEV_PER_DRIVE)
#define SUB_PER_DRIVE   (NR_PARTITIONS * NR_PARTITIONS)
#define NR_SUBDEVS      (MAX_DRIVES * SUB_PER_DRIVE)
#define TIMEOUT        32000    /* controller timeout in ms */
#define RECOVERYTIME     500    /* controller recovery time in ms */
#define INITIALIZED     0x01    /* drive is initialized */
#define DEAF            0x02    /* controller must be reset */
```

```
#define SMART           0x04    /* drive supports ATA commands */


/* Variables. */
PRIVATE struct wini {           /* main drive struct, one entry per drive */
  unsigned state;               /* drive state: deaf, initialized, dead */
  unsigned base;                /* base register of the register file */
  unsigned irq;                 /* interrupt request line */
  unsigned lcylinders;          /* logical number of cylinders (BIOS) */
  unsigned lheads;              /* logical number of heads */
  unsigned lsectors;            /* logical number of sectors per track */
  unsigned pcylinders;          /* physical number of cylinders (translated) */
  unsigned pheads;              /* physical number of heads */
  unsigned psectors;            /* physical number of sectors per track */
  unsigned ldhpref;             /* top four bytes of the LDH (head) register */
  unsigned precomp;             /* write precompensation cylinder / 4 */
  unsigned max_count;           /* max request for this drive */
  unsigned open_ct;             /* in-use count */
  struct device part[DEV_PER_DRIVE];    /* primary partitions: hd[0-4] */
  struct device subpart[SUB_PER_DRIVE]; /* subpartitions: hd[1-4][a-d] */
} wini[MAX_DRIVES], *w_wn;

PRIVATE struct trans {
  struct iorequest_s *iop;      /* belongs to this I/O request */
  unsigned long block;          /* first sector to transfer */
  unsigned count;               /* byte count */
  phys_bytes phys;              /* user physical address */
} wtrans[NR_IOREQS];

PRIVATE struct trans *w_tp;             /* to add transfer requests */
PRIVATE unsigned w_count;               /* number of bytes to transfer */
PRIVATE unsigned long w_nextblock;      /* next block on disk to transfer */
PRIVATE int w_opcode;                   /* DEV_READ or DEV_WRITE */
PRIVATE int w_command;                  /* current command in execution */
PRIVATE int w_status;                   /* status after interrupt */
PRIVATE int w_drive;                    /* selected drive */
PRIVATE struct device *w_dv;            /* device's base and size */

FORWARD _PROTOTYPE( void init_params, (void) );
FORWARD _PROTOTYPE( int w_do_open, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( struct device *w_prepare, (int device) );
FORWARD _PROTOTYPE( int w_identify, (void) );
FORWARD _PROTOTYPE( char *w_name, (void) );
FORWARD _PROTOTYPE( int w_specify, (void) );
FORWARD _PROTOTYPE( int w_schedule, (int proc_nr, struct iorequest_s *iop) );
FORWARD _PROTOTYPE( int w_finish, (void) );
FORWARD _PROTOTYPE( int com_out, (struct command *cmd) );
FORWARD _PROTOTYPE( void w_need_reset, (void) );
FORWARD _PROTOTYPE( int w_do_close, (struct driver *dp, message *m_ptr) );
FORWARD _PROTOTYPE( int com_simple, (struct command *cmd) );
FORWARD _PROTOTYPE( void w_timeout, (void) );
FORWARD _PROTOTYPE( int w_reset, (void) );
FORWARD _PROTOTYPE( int w_intr_wait, (void) );
FORWARD _PROTOTYPE( int w_waitfor, (int mask, int value) );
FORWARD _PROTOTYPE( int w_handler, (int irq) );
FORWARD _PROTOTYPE( void w_geometry, (struct partition *entry) );

/* w_waitfor loop unrolled once for speed. */
#define waitfor(mask, value)    \
        ((in_byte(w_wn->base + REG_STATUS) & mask) == value \
```

```
                || w_waitfor(mask, value))


/* Entry points to this driver. */
PRIVATE struct driver w_dtab = {
  w_name,               /* current device's name */
  w_do_open,            /* open or mount request, initialize device */
  w_do_close,           /* release device */
  do_diocntl,           /* get or set a partition's geometry */
  w_prepare,            /* prepare for I/O on a given minor device */
  w_schedule,           /* precompute cylinder, head, sector, etc. */
  w_finish,             /* do the I/O */
  nop_cleanup,          /* nothing to clean up */
  w_geometry,           /* tell the geometry of the disk */
};

#if ENABLE_ATAPI
#include "atapi.c"      /* extra code for ATAPI CD-ROM */
#endif


/*===========================================================================*
 *                              at_winchester_task                           *
 *===========================================================================*/
PUBLIC void at_winchester_task()
{
/* Set special disk parameters then call the generic main loop. */

  init_params();

  driver_task(&w_dtab);
}


/*============================================================================*
 *                              init_params                                   *
 *============================================================================*/
PRIVATE void init_params()
{
/* This routine is called at startup to initialize the drive parameters. */

  u16_t parv[2];
  unsigned int vector;
  int drive, nr_drives, i;
  struct wini *wn;
  u8_t params[16];
  phys_bytes param_phys = vir2phys(params);

  /* Get the number of drives from the BIOS data area */
  phys_copy(0x475L, param_phys, 1L);
  if ((nr_drives = params[0]) > 2) nr_drives = 2;

  for (drive = 0, wn = wini; drive < MAX_DRIVES; drive++, wn++) {
        if (drive < nr_drives) {
                /* Copy the BIOS parameter vector */
                vector = drive == 0 ? WINI_0_PARM_VEC : WINI_1_PARM_VEC;
                phys_copy(vector * 4L, vir2phys(parv), 4L);

                /* Calculate the address of the parameters and copy them */
                phys_copy(hclick_to_physb(parv[1]) + parv[0], param_phys, 16L);
```

```

                /* Copy the parameters to the structures of the drive */
                wn->lcylinders = bp_cylinders(params);
                wn->lheads = bp_heads(params);
                wn->lsectors = bp_sectors(params);
                wn->precomp = bp_precomp(params) >> 2;
        }
        wn->ldhpref = ldh_init(drive);
        wn->max_count = MAX_SECS << SECTOR_SHIFT;
        if (drive < 2) {
                /* Controller 0. */
                wn->base = REG_BASE0;
                wn->irq = AT_IRQ0;
        } else {
                /* Controller 1. */
                wn->base = REG_BASE1;
                wn->irq = AT_IRQ1;
        }
  }
}


/*===========================================================================*
 *                              w_do_open                                    *
 *===========================================================================*/
PRIVATE int w_do_open(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Device open: Initialize the controller and read the partition table. */

  int r;
  struct wini *wn;
  struct command cmd;

  if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
  wn = w_wn;

  if (wn->state == 0) {
        /* Try to identify the device. */
        if (w_identify() != OK) {
                printf("%s: probe failed\n", w_name());
                if (wn->state & DEAF) w_reset();
                wn->state = 0;
                return(ENXIO);
        }
  }
  if (wn->open_ct++ == 0) {
        /* Partition the disk. */
        partition(&w_dtab, w_drive * DEV_PER_DRIVE, P_PRIMARY);
  }
  return(OK);
}


/*==========================================================================*
 *                              w_prepare                                   *
 *==========================================================================*/
PRIVATE struct device *w_prepare(device)
int device;
```

```
{
/* Prepare for I/O on a device. */

  /* Nothing to transfer as yet. */
  w_count = 0;

  if (device < NR_DEVICES) {                      /* hd0, hd1, ... */
        w_drive = device / DEV_PER_DRIVE;       /* save drive number */
        w_wn = &wini[w_drive];
        w_dv = &w_wn->part[device % DEV_PER_DRIVE];
  } else
  if ((unsigned) (device -= MINOR_hd1a) < NR_SUBDEVS) { /* hd1a, hd1b, ... */
        w_drive = device / SUB_PER_DRIVE;
        w_wn = &wini[w_drive];
        w_dv = &w_wn->subpart[device % SUB_PER_DRIVE];
  } else {
        return(NIL_DEV);
  }
  return(w_dv);
}


/*===========================================================================*
 *                              w_identify                                   *
 *===========================================================================*/
PRIVATE int w_identify()
{
/* Find out if a device exists, if it is an old AT disk, or a newer ATA
 * drive, a removable media device, etc.
 */

  struct wini *wn = w_wn;
  struct command cmd;
  char id_string[40];
  int i, r;
  unsigned long size;
#define id_byte(n)      (&tmp_buf[2 * (n)])
#define id_word(n)      (((u16_t) id_byte(n)[0] <<  0) \
                        |((u16_t) id_byte(n)[1] <<  8))
#define id_longword(n)  (((u32_t) id_byte(n)[0] <<  0) \
                        |((u32_t) id_byte(n)[1] <<  8) \
                        |((u32_t) id_byte(n)[2] << 16) \
                        |((u32_t) id_byte(n)[3] << 24))

  /* Check if the one of the registers exists. */
  r = in_byte(wn->base + REG_CYL_LO);
  out_byte(wn->base + REG_CYL_LO, ~r);
  if (in_byte(wn->base + REG_CYL_LO) == r) return(ERR);

  /* Looks OK; register IRQ and try an ATA identify command. */
  put_irq_handler(wn->irq, w_handler);
  enable_irq(wn->irq);

  cmd.ldh     = wn->ldhpref;
  cmd.command = ATA_IDENTIFY;
  if (com_simple(&cmd) == OK) {
        /* This is an ATA device. */
        wn->state |= SMART;

        /* Device information. */
```

```
                port_read(wn->base + REG_DATA, tmp_phys, SECTOR_SIZE);

                /* Why are the strings byte swapped??? */
                for (i = 0; i < 40; i++) id_string[i] = id_byte(27)[i^1];

                /* Preferred CHS translation mode. */
                wn->pcylinders = id_word(1);
                wn->pheads = id_word(3);
                wn->psectors = id_word(6);
                size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;

                if ((id_byte(49)[1] & 0x02) && size > 512L*1024*2) {
                        /* Drive is LBA capable and is big enough to trust it to
                         * not make a mess of it.
                         */
                        wn->ldhpref |= LDH_LBA;
                        size = id_longword(60);
                }

                if (wn->lcylinders == 0) {
                        /* No BIOS parameters?  Then make some up. */
                        wn->lcylinders = wn->pcylinders;
                        wn->lheads = wn->pheads;
                        wn->lsectors = wn->psectors;
                        while (wn->lcylinders > 1024) {
                                wn->lheads *= 2;
                                wn->lcylinders /= 2;
                        }
                }
        } else {
                /* Not an ATA device; no translations, no special features.  Don't
                 * touch it unless the BIOS knows about it.
                 */
                if (wn->lcylinders == 0) return(ERR);   /* no BIOS parameters */
                wn->pcylinders = wn->lcylinders;
                wn->pheads = wn->lheads;
                wn->psectors = wn->lsectors;
                size = (u32_t) wn->pcylinders * wn->pheads * wn->psectors;
        }
        /* The fun ends at 4 GB. */
        if (size > ((u32_t) -1) / SECTOR_SIZE) size = ((u32_t) -1) / SECTOR_SIZE;

        /* Base and size of the whole drive */
        wn->part[0].dv_base = 0;
        wn->part[0].dv_size = size << SECTOR_SHIFT;

        if (w_specify() != OK && w_specify() != OK) return(ERR);

        printf("%s: ", w_name());
        if (wn->state & SMART) {
                printf("%.40s\n", id_string);
        } else {
                printf("%ux%ux%u\n", wn->pcylinders, wn->pheads, wn->psectors);
        }
        return(OK);
}


/*===========================================================================*
 *                              w_name                                       *
```

```
 *===========================================================================*/
PRIVATE char *w_name()
{
/* Return a name for the current device. */
  static char name[] = "at-hd15";
  unsigned device = w_drive * DEV_PER_DRIVE;

  if (device < 10) {
        name[5] = '0' + device;
        name[6] = 0;
  } else {
        name[5] = '0' + device / 10;
        name[6] = '0' + device % 10;
  }
  return name;
}


/*===========================================================================*
 *                              w_specify                                    *
 *===========================================================================*/
PRIVATE int w_specify()
{
/* Routine to initialize the drive after boot or when a reset is needed. */

  struct wini *wn = w_wn;
  struct command cmd;

  if ((wn->state & DEAF) && w_reset() != OK) return(ERR);

  /* Specify parameters: precompensation, number of heads and sectors. */
  cmd.precomp = wn->precomp;
  cmd.count   = wn->psectors;
  cmd.ldh     = w_wn->ldhpref | (wn->pheads - 1);
  cmd.command = CMD_SPECIFY;            /* Specify some parameters */

  if (com_simple(&cmd) != OK) return(ERR);

  if (!(wn->state & SMART)) {
        /* Calibrate an old disk. */
        cmd.sector  = 0;
        cmd.cyl_lo  = 0;
        cmd.cyl_hi  = 0;
        cmd.ldh     = w_wn->ldhpref;
        cmd.command = CMD_RECALIBRATE;

        if (com_simple(&cmd) != OK) return(ERR);
  }

  wn->state |= INITIALIZED;
  return(OK);
}


/*===========================================================================*
 *                              w_schedule                                   *
 *===========================================================================*/
PRIVATE int w_schedule(proc_nr, iop)
int proc_nr;                    /* process doing the request */
struct iorequest_s *iop;        /* pointer to read or write request */
```

```
{
/* Gather I/O requests on consecutive blocks so they may be read/written
 * in one controller command.  (There is enough time to compute the next
 * consecutive request while an unwanted block passes by.)
 */
  struct wini *wn = w_wn;
  int r, opcode;
  unsigned long pos;
  unsigned nbytes, count;
  unsigned long block;
  phys_bytes user_phys;

  /* This many bytes to read/write */
  nbytes = iop->io_nbytes;
  if ((nbytes & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);

  /* From/to this position on the device */
  pos = iop->io_position;
  if ((pos & SECTOR_MASK) != 0) return(iop->io_nbytes = EINVAL);

  /* To/from this user address */
  user_phys = numap(proc_nr, (vir_bytes) iop->io_buf, nbytes);
  if (user_phys == 0) return(iop->io_nbytes = EINVAL);

  /* Read or write? */
  opcode = iop->io_request & ~OPTIONAL_IO;

  /* Which block on disk and how close to EOF? */
  if (pos >= w_dv->dv_size) return(OK);         /* At EOF */
  if (pos + nbytes > w_dv->dv_size) nbytes = w_dv->dv_size - pos;
  block = (w_dv->dv_base + pos) >> SECTOR_SHIFT;

  if (w_count > 0 && block != w_nextblock) {
        /* This new request can't be chained to the job being built */
        if ((r = w_finish()) != OK) return(r);
  }

  /* The next consecutive block */
  w_nextblock = block + (nbytes >> SECTOR_SHIFT);

  /* While there are "unscheduled" bytes in the request: */
  do {
        count = nbytes;

        if (w_count == wn->max_count) {
                /* The drive can't do more then max_count at once */
                if ((r = w_finish()) != OK) return(r);
        }

        if (w_count + count > wn->max_count)
                count = wn->max_count - w_count;

        if (w_count == 0) {
                /* The first request in a row, initialize. */
                w_opcode = opcode;
                w_tp = wtrans;
        }

        /* Store I/O parameters */
        w_tp->iop = iop;
```

```
        w_tp->block = block;
        w_tp->count = count;
        w_tp->phys = user_phys;

        /* Update counters */
        w_tp++;
        w_count += count;
        block += count >> SECTOR_SHIFT;
        user_phys += count;
        nbytes -= count;
  } while (nbytes > 0);

  return(OK);
}


/*===========================================================================*
 *                              w_finish                                     *
 *===========================================================================*/
PRIVATE int w_finish()
{
/* Carry out the I/O requests gathered in wtrans[]. */

  struct trans *tp = wtrans;
  struct wini *wn = w_wn;
  int r, errors;
  struct command cmd;
  unsigned cylinder, head, sector, secspcyl;

  if (w_count == 0) return(OK); /* Spurious finish. */

  r = ERR;      /* Trigger the first com_out */
  errors = 0;

  do {
        if (r != OK) {
                /* The controller must be (re)programmed. */

                /* First check to see if a reinitialization is needed. */
                if (!(wn->state & INITIALIZED) && w_specify() != OK)
                        return(tp->iop->io_nbytes = EIO);

                /* Tell the controller to transfer w_count bytes */
                cmd.precomp = wn->precomp;
                cmd.count   = (w_count >> SECTOR_SHIFT) & BYTE;
                if (wn->ldhpref & LDH_LBA) {
                        cmd.sector  = (tp->block >>  0) & 0xFF;
                        cmd.cyl_lo  = (tp->block >>  8) & 0xFF;
                        cmd.cyl_hi  = (tp->block >> 16) & 0xFF;
                        cmd.ldh     = wn->ldhpref | ((tp->block >> 24) & 0xF);
                } else {
                        secspcyl = wn->pheads * wn->psectors;
                        cylinder = tp->block / secspcyl;
                        head = (tp->block % secspcyl) / wn->psectors;
                        sector = tp->block % wn->psectors;
                        cmd.sector  = sector + 1;
                        cmd.cyl_lo  = cylinder & BYTE;
                        cmd.cyl_hi  = (cylinder >> 8) & BYTE;
                        cmd.ldh     = wn->ldhpref | head;
                }
```

```
                cmd.command = w_opcode == DEV_WRITE ? CMD_WRITE : CMD_READ;

                if ((r = com_out(&cmd)) != OK) {
                        if (++errors == MAX_ERRORS) {
                                w_command = CMD_IDLE;
                                return(tp->iop->io_nbytes = EIO);
                        }
                        continue;       /* Retry */
                }
        }

        /* For each sector, wait for an interrupt and fetch the data (read),
         * or supply data to the controller and wait for an interrupt (write).
         */

        if (w_opcode == DEV_READ) {
                if ((r = w_intr_wait()) == OK) {
                        /* Copy data from the device's buffer to user space. */

                        port_read(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);

                        tp->phys += SECTOR_SIZE;
                        tp->iop->io_nbytes -= SECTOR_SIZE;
                        w_count -= SECTOR_SIZE;
                        if ((tp->count -= SECTOR_SIZE) == 0) tp++;
                } else {
                        /* Any faulty data? */
                        if (w_status & STATUS_DRQ) {
                                port_read(wn->base + REG_DATA, tmp_phys,
                                                                SECTOR_SIZE);
                        }
                }
        } else {
                /* Wait for data requested. */
                if (!waitfor(STATUS_DRQ, STATUS_DRQ)) {
                        r = ERR;
                } else {
                        /* Fill the buffer of the drive. */

                        port_write(wn->base + REG_DATA, tp->phys, SECTOR_SIZE);
                        r = w_intr_wait();
                }

                if (r == OK) {
                        /* Book the bytes successfully written. */

                        tp->phys += SECTOR_SIZE;
                        tp->iop->io_nbytes -= SECTOR_SIZE;
                        w_count -= SECTOR_SIZE;
                        if ((tp->count -= SECTOR_SIZE) == 0) tp++;
                }
        }

        if (r != OK) {
                /* Don't retry if sector marked bad or too many errors */
                if (r == ERR_BAD_SECTOR || ++errors == MAX_ERRORS) {
                        w_command = CMD_IDLE;
                        return(tp->iop->io_nbytes = EIO);
                }

```

```
                /* Reset if halfway, but bail out if optional I/O. */
                if (errors == MAX_ERRORS / 2) {
                        w_need_reset();
                        if (tp->iop->io_request & OPTIONAL_IO) {
                                w_command = CMD_IDLE;
                                return(tp->iop->io_nbytes = EIO);
                        }
                }
                continue;       /* Retry */
        }
        errors = 0;
  } while (w_count > 0);

  w_command = CMD_IDLE;
  return(OK);
}


/*===========================================================================*
 *                              com_out                                      *
 *===========================================================================*/
PRIVATE int com_out(cmd)
struct command *cmd;            /* Command block */
{
/* Output the command block to the winchester controller and return status */

  struct wini *wn = w_wn;
  unsigned base = wn->base;

  if (!waitfor(STATUS_BSY, 0)) {
        printf("%s: controller not ready\n", w_name());
        return(ERR);
  }

  /* Select drive. */
  out_byte(base + REG_LDH, cmd->ldh);

  if (!waitfor(STATUS_BSY, 0)) {
        printf("%s: drive not ready\n", w_name());
        return(ERR);
  }

  /* Schedule a wakeup call, some controllers are flaky. */
  clock_mess(WAKEUP, w_timeout);

  out_byte(base + REG_CTL, wn->pheads >= 8 ? CTL_EIGHTHEADS : 0);
  out_byte(base + REG_PRECOMP, cmd->precomp);
  out_byte(base + REG_COUNT, cmd->count);
  out_byte(base + REG_SECTOR, cmd->sector);
  out_byte(base + REG_CYL_LO, cmd->cyl_lo);
  out_byte(base + REG_CYL_HI, cmd->cyl_hi);
  lock();
  out_byte(base + REG_COMMAND, cmd->command);
  w_command = cmd->command;
  w_status = STATUS_BSY;
  unlock();
  return(OK);
}
```

```
/*===========================================================================*
 *                              w_need_reset                                 *
 *===========================================================================*/
PRIVATE void w_need_reset()
{
/* The controller needs to be reset. */
  struct wini *wn;

  for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
        wn->state |= DEAF;
        wn->state &= ~INITIALIZED;
  }
}


/*===========================================================================*
 *                              w_do_close                                   *
 *===========================================================================*/
PRIVATE int w_do_close(dp, m_ptr)
struct driver *dp;
message *m_ptr;
{
/* Device close: Release a device. */

  if (w_prepare(m_ptr->DEVICE) == NIL_DEV) return(ENXIO);
  w_wn->open_ct--;
  return(OK);
}


/*===========================================================================*
 *                              com_simple                                   *
 *===========================================================================*/
PRIVATE int com_simple(cmd)
struct command *cmd;            /* Command block */
{
/* A simple controller command, only one interrupt and no data-out phase. */
  int r;

  if ((r = com_out(cmd)) == OK) r = w_intr_wait();
  w_command = CMD_IDLE;
  return(r);
}


/*===========================================================================*
 *                              w_timeout                                    *
 *===========================================================================*/
PRIVATE void w_timeout()
{
  struct wini *wn = w_wn;

  switch (w_command) {
  case CMD_IDLE:
        break;          /* fine */
  case CMD_READ:
  case CMD_WRITE:
        /* Impossible, but not on PC's:  The controller does not respond. */

        /* Limiting multisector I/O seems to help. */
```

```
        if (wn->max_count > 8 * SECTOR_SIZE) {
                wn->max_count = 8 * SECTOR_SIZE;
        } else {
                wn->max_count = SECTOR_SIZE;
        }
        /*FALL THROUGH*/
  default:
        /* Some other command. */
        printf("%s: timeout on command %02x\n", w_name(), w_command);
        w_need_reset();
        w_status = 0;
        interrupt(WINCHESTER);
  }
}


/*===========================================================================*
 *                              w_reset                                      *
 *===========================================================================*/
PRIVATE int w_reset()
{
/* Issue a reset to the controller.  This is done after any catastrophe,
 * like the controller refusing to respond.
 */

  struct wini *wn;
  int err;

  /* Wait for any internal drive recovery. */
  milli_delay(RECOVERYTIME);

  /* Strobe reset bit */
  out_byte(w_wn->base + REG_CTL, CTL_RESET);
  milli_delay(1);
  out_byte(w_wn->base + REG_CTL, 0);
  milli_delay(1);

  /* Wait for controller ready */
  if (!w_waitfor(STATUS_BSY | STATUS_RDY, STATUS_RDY)) {
        printf("%s: reset failed, drive busy\n", w_name());
        return(ERR);
  }

  /* The error register should be checked now, but some drives mess it up. */

  for (wn = wini; wn < &wini[MAX_DRIVES]; wn++) {
        if (wn->base == w_wn->base) wn->state &= ~DEAF;
  }
  return(OK);
}


/*============================================================================*
 *                              w_intr_wait                                   *
 *============================================================================*/
PRIVATE int w_intr_wait()
{
/* Wait for a task completion interrupt and return results. */

  message mess;
```

```
  int r;

  /* Wait for an interrupt that sets w_status to "not busy". */
  while (w_status & STATUS_BSY) receive(HARDWARE, &mess);

  /* Check status. */
  lock();
  if ((w_status & (STATUS_BSY | STATUS_RDY | STATUS_WF | STATUS_ERR))
                                                        == STATUS_RDY) {
        r = OK;
        w_status |= STATUS_BSY; /* assume still busy with I/O */
  } else
  if ((w_status & STATUS_ERR) && (in_byte(w_wn->base + REG_ERROR) & ERROR_BB)) {
        r = ERR_BAD_SECTOR;     /* sector marked bad, retries won't help */
  } else {
        r = ERR;                /* any other error */
  }
  unlock();
  return(r);
}


/*===========================================================================*
 *                              w_waitfor                                    *
 *===========================================================================*/
PRIVATE int w_waitfor(mask, value)
int mask;                       /* status mask */
int value;                      /* required status */
{
/* Wait until controller is in the required state.  Return zero on timeout. */

  struct milli_state ms;

  milli_start(&ms);
  do {
       if ((in_byte(w_wn->base + REG_STATUS) & mask) == value) return 1;
  } while (milli_elapsed(&ms) < TIMEOUT);

  w_need_reset();       /* Controller gone deaf. */
  return(0);
}


/*===========================================================================*
 *                              w_handler                                    *
 *===========================================================================*/
PRIVATE int w_handler(irq)
int irq;
{
/* Disk interrupt, send message to winchester task and reenable interrupts. */

  w_status = in_byte(w_wn->base + REG_STATUS);  /* acknowledge interrupt */
  interrupt(WINCHESTER);
  return 1;
}


/*=============================================================================*
 *                              w_geometry                                     *
 *=============================================================================*/
```

```
PRIVATE void w_geometry(entry)
struct partition *entry;
{
  entry->cylinders = w_wn->lcylinders;
  entry->heads = w_wn->lheads;
  entry->sectors = w_wn->lsectors;
}
#endif /* ENABLE_AT_WINI */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/clock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the code and data for the clock task.  The clock task
 * accepts six message types:
 *
 *   HARD_INT:    a clock interrupt has occurred
 *   GET_UPTIME:  get the time since boot in ticks
 *   GET_TIME:    a process wants the real time in seconds
 *   SET_TIME:    a process wants to set the real time in seconds
 *   SET_ALARM:   a process wants to be alerted after a specified interval
 *   SET_SYN_AL:  set the sync alarm
 *
 *
 * The input message is format m6.  The parameters are as follows:
 *
 *     m_type    CLOCK_PROC   FUNC    NEW_TIME
 * ---------------------------------------------
 * | HARD_INT   |          |         |         |
 * |------------+----------+---------+---------|
 * | GET_UPTIME |          |         |         |
 * |------------+----------+---------+---------|
 * | GET_TIME   |          |         |         |
 * |------------+----------+---------+---------|
 * | SET_TIME   |          |         | newtime |
 * |------------+----------+---------+---------|
 * | SET_ALARM  | proc_nr  |f to call|  delta  |
 * |------------+----------+---------+---------|
 * | SET_SYN_AL | proc_nr  |         |  delta  |
 * ---------------------------------------------
 * NEW_TIME, DELTA_CLICKS, and SECONDS_LEFT all refer to the same field in
 * the message, depending upon the message type.
 *
 * Reply messages are of type OK, except in the case of a HARD_INT, to
 * which no reply is generated. For the GET_* messages the time is returned
 * in the NEW_TIME field, and for the SET_ALARM and SET_SYN_AL the time
 * in seconds remaining until the alarm is returned is returned in the same
 * field.
 *
 * When an alarm goes off, if the caller is a user process, a SIGALRM signal
 * is sent to it.  If it is a task, a function specified by the caller will
 * be invoked.  This function may, for example, send a message, but only if
 * it is certain that the task will be blocked when the timer goes off. A
 * synchronous alarm sends a message to the synchronous alarm task, which
 * in turn can dispatch a message to another server. This is the only way
 * to send an alarm to a server, since servers cannot use the function-call
 * mechanism available to tasks and servers cannot receive signals.
 */
```

```

#include "kernel.h"
#include <signal.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"

/* Constant definitions. */
#define MILLISEC         100    /* how often to call the scheduler (msec) */
#define SCHED_RATE (MILLISEC*HZ/1000)   /* number of ticks per schedule */

/* Clock parameters. */
#define COUNTER_FREQ (2*TIMER_FREQ)     /* counter frequency using sqare wave*/
#define LATCH_COUNT     0x00    /* cc00xxxx, c = channel, x = any */
#define SQUARE_WAVE     0x36    /* ccaammmb, a = access, m = mode, b = BCD */
                                /*   11x11, 11 = LSB then MSB, x11 = sq wave */
#define TIMER_COUNT ((unsigned) (TIMER_FREQ/HZ)) /* initial value for counter*/
#define TIMER_FREQ  1193182L    /* clock frequency for timer in PC and AT */

#define CLOCK_ACK_BIT   0x80    /* PS/2 clock interrupt acknowledge bit */

/* Clock task variables. */
PRIVATE clock_t realtime;       /* real time clock */
PRIVATE time_t boot_time;       /* time in seconds of system boot */
PRIVATE clock_t next_alarm;     /* probable time of next alarm */
PRIVATE message mc;             /* message buffer for both input and output */
PRIVATE int watchdog_proc;      /* contains proc_nr at call of *watch_dog[]*/
PRIVATE watchdog_t watch_dog[NR_TASKS+NR_PROCS];

/* Variables used by both clock task and synchronous alarm task */
PRIVATE int syn_al_alive= TRUE; /* don’t wake syn_alrm_task before inited*/
PRIVATE int syn_table[NR_TASKS+NR_PROCS]; /* which tasks get CLOCK_INT*/

/* Variables changed by interrupt handler */
PRIVATE clock_t pending_ticks;  /* ticks seen by low level only */
PRIVATE int sched_ticks = SCHED_RATE;   /* counter: when 0, call scheduler */
PRIVATE struct proc *prev_ptr;  /* last user process run by clock task */

FORWARD _PROTOTYPE( void common_setalarm, (int proc_nr,
                long delta_ticks, watchdog_t fuction) );
FORWARD _PROTOTYPE( void do_clocktick, (void) );
FORWARD _PROTOTYPE( void do_get_time, (void) );
FORWARD _PROTOTYPE( void do_getuptime, (void) );
FORWARD _PROTOTYPE( void do_set_time, (message *m_ptr) );
FORWARD _PROTOTYPE( void do_setalarm, (message *m_ptr) );
FORWARD _PROTOTYPE( void init_clock, (void) );
FORWARD _PROTOTYPE( void cause_alarm, (void) );
FORWARD _PROTOTYPE( void do_setsyn_alrm, (message *m_ptr) );
FORWARD _PROTOTYPE( int clock_handler, (int irq) );

/*===========================================================================*
 *                              clock_task                                   *
 *===========================================================================*/
PUBLIC void clock_task()
{
/* Main program of clock task.  It corrects realtime by adding pending
 * ticks seen only by the interrupt service, then it determines which
 * of the 6 possible calls this is by looking at ’mc.m_type’.  Then
 * it dispatches.
 */
```

```
  int opcode;

  init_clock();                 /* initialize clock task */

  /* Main loop of the clock task.  Get work, process it, sometimes reply. */
  while (TRUE) {
     receive(ANY, &mc);         /* go get a message */
     opcode = mc.m_type;        /* extract the function code */

     lock();
     realtime += pending_ticks; /* transfer ticks from low level handler */
     pending_ticks = 0;         /* so we don't have to worry about them */
     unlock();

     switch (opcode) {
        case HARD_INT:   do_clocktick();        break;
        case GET_UPTIME: do_getuptime();        break;
        case GET_TIME:   do_get_time();         break;
        case SET_TIME:   do_set_time(&mc);      break;
        case SET_ALARM:  do_setalarm(&mc);      break;
        case SET_SYNC_AL:do_setsyn_alrm(&mc);   break;
        default: panic("clock task got bad message", mc.m_type);
     }

    /* Send reply, except for clock tick. */
    mc.m_type = OK;
    if (opcode != HARD_INT) send(mc.m_source, &mc);
  }
}


/*===========================================================================*
 *                              do_clocktick                                 *
 *===========================================================================*/
PRIVATE void do_clocktick()
{
/* Despite its name, this routine is not called on every clock tick. It
 * is called on those clock ticks when a lot of work needs to be done.
 */

  register struct proc *rp;
  register int proc_nr;

  if (next_alarm <= realtime) {
        /* An alarm may have gone off, but proc may have exited, so check. */
        next_alarm = LONG_MAX;  /* start computing next alarm */
        for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
                if (rp->p_alarm != 0) {
                        /* See if this alarm time has been reached. */
                        if (rp->p_alarm <= realtime) {
                                /* A timer has gone off.  If it is a user proc,
                                 * send it a signal.  If it is a task, call the
                                 * function previously specified by the task.
                                 */
                                proc_nr = proc_number(rp);
                                if (watch_dog[proc_nr+NR_TASKS]) {
                                        watchdog_proc= proc_nr;
                                        (*watch_dog[proc_nr+NR_TASKS])();
                                }
```

```
                                else
                                        cause_sig(proc_nr, SIGALRM);
                                rp->p_alarm = 0;
                        }

                        /* Work on determining which alarm is next. */
                        if (rp->p_alarm != 0 && rp->p_alarm < next_alarm)
                                next_alarm = rp->p_alarm;
                }
        }
  }

  /* If a user process has been running too long, pick another one. */
  if (--sched_ticks == 0) {
        if (bill_ptr == prev_ptr) lock_sched(); /* process has run too long */
        sched_ticks = SCHED_RATE;               /* reset quantum */
        prev_ptr = bill_ptr;                    /* new previous process */
  }
}


/*===========================================================================*
 *                              do_getuptime                                 *
 *===========================================================================*/
PRIVATE void do_getuptime()
{
/* Get and return the current clock uptime in ticks. */

  mc.NEW_TIME = realtime;       /* current uptime */
}


/*===========================================================================*
 *                              get_uptime                                   *
 *===========================================================================*/
PUBLIC clock_t get_uptime()
{
/* Get and return the current clock uptime in ticks.  This function is
 * designed to be called from other tasks, so they can get uptime without
 * the overhead of messages. It has to be careful about pending_ticks.
 */

  clock_t uptime;

  lock();
  uptime = realtime + pending_ticks;
  unlock();
  return(uptime);
}


/*===========================================================================*
 *                              do_get_time                                  *
 *===========================================================================*/
PRIVATE void do_get_time()
{
/* Get and return the current clock time in seconds. */

  mc.NEW_TIME = boot_time + realtime/HZ;        /* current real time */
}
```

```c
/*===========================================================================*
 *                              do_set_time                                  *
 *===========================================================================*/
PRIVATE void do_set_time(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Set the real time clock.  Only the superuser can use this call. */

  boot_time = m_ptr->NEW_TIME - realtime/HZ;
}


/*===========================================================================*
 *                              do_setalarm                                  *
 *===========================================================================*/
PRIVATE void do_setalarm(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* A process wants an alarm signal or a task wants a given watch_dog function
 * called after a specified interval.
 */

  register struct proc *rp;
  int proc_nr;                  /* which process wants the alarm */
  long delta_ticks;             /* in how many clock ticks does he want it? */
  watchdog_t function;          /* function to call (tasks only) */

  /* Extract the parameters from the message. */
  proc_nr = m_ptr->CLOCK_PROC_NR;       /* process to interrupt later */
  delta_ticks = m_ptr->DELTA_TICKS;     /* how many ticks to wait */
  function = (watchdog_t) m_ptr->FUNC_TO_CALL;
                                        /* function to call (tasks only) */
  rp = proc_addr(proc_nr);
  mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm - realtime)/HZ );
  if (!istaskp(rp)) function= 0;        /* user processes get signaled */
  common_setalarm(proc_nr, delta_ticks, function);
}


/*===========================================================================*
 *                              do_setsyn_alrm                               *
 *===========================================================================*/
PRIVATE void do_setsyn_alrm(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* A process wants a synchronous alarm.
 */

  register struct proc *rp;
  int proc_nr;                  /* which process wants the alarm */
  long delta_ticks;             /* in how many clock ticks does he want it? */

  /* Extract the parameters from the message. */
  proc_nr = m_ptr->CLOCK_PROC_NR;       /* process to interrupt later */
  delta_ticks = m_ptr->DELTA_TICKS;     /* how many ticks to wait */
  rp = proc_addr(proc_nr);
  mc.SECONDS_LEFT = (rp->p_alarm == 0 ? 0 : (rp->p_alarm - realtime)/HZ );
  common_setalarm(proc_nr, delta_ticks, cause_alarm);
```

```c
}


/*===========================================================================*
 *                              common_setalarm                              *
 *===========================================================================*/
PRIVATE void common_setalarm(proc_nr, delta_ticks, function)
int proc_nr;                    /* which process wants the alarm */
long delta_ticks;               /* in how many clock ticks does he want it? */
watchdog_t function;            /* function to call (0 if cause_sig is
                                 * to be called */
{
/* Finish up work of do_set_alarm and do_setsyn_alrm.  Record an alarm
 * request and check to see if it is the next alarm needed.
 */

  register struct proc *rp;

  rp = proc_addr(proc_nr);
  rp->p_alarm = (delta_ticks == 0 ? 0 : realtime + delta_ticks);
  watch_dog[proc_nr+NR_TASKS] = function;

  /* Which alarm is next? */
  next_alarm = LONG_MAX;
  for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++)
        if(rp->p_alarm != 0 && rp->p_alarm < next_alarm)next_alarm=rp->p_alarm;

}


/*===========================================================================*
 *                              cause_alarm                                  *
 *===========================================================================*/
PRIVATE void cause_alarm()
{
/* Routine called if a timer goes off and the process requested a synchronous
 * alarm. The process number is in the global variable watchdog_proc (HACK).
 */
  message mess;

  syn_table[watchdog_proc + NR_TASKS]= TRUE;
  if (!syn_al_alive) send (SYN_ALRM_TASK, &mess);
}


/*===========================================================================*
 *                              syn_alrm_task                                *
 *===========================================================================*/
PUBLIC void syn_alrm_task()
{
/* Main program of the synchronous alarm task.
 * This task receives messages only from cause_alarm in the clock task.
 * It sends a CLOCK_INT message to a process that requested a syn_alrm.
 * Synchronous alarms are so called because, unlike a signals or the
 * activation of a watchdog, a synchronous alarm is received by a process
 * when it is in a known part of its code, that is, when it has issued
 * a call to receive a message.
 */

  message mess;
```

```
  int work_done;        /* ready to sleep ? */
  int *al_ptr;          /* pointer in syn_table */
  int i;

  syn_al_alive= TRUE;
  for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
        *al_ptr= FALSE;

  while (TRUE) {
        work_done= TRUE;
        for (i= 0, al_ptr= syn_table; i<NR_TASKS+NR_PROCS; i++, al_ptr++)
                if (*al_ptr) {
                        *al_ptr= FALSE;
                        mess.m_type= CLOCK_INT;
                        send (i-NR_TASKS, &mess);
                        work_done= FALSE;
                }
        if (work_done) {
                syn_al_alive= FALSE;
                receive (CLOCK, &mess);
                syn_al_alive= TRUE;
        }
  }
}


/*===========================================================================*
 *                              clock_handler                                *
 *===========================================================================*/
PRIVATE int clock_handler(irq)
int irq;
{
/* This executes on every clock tick (i.e., every time the timer chip
 * generates an interrupt). It does a little bit of work so the clock
 * task does not have to be called on every tick.
 *
 * Switch context to do_clocktick if an alarm has gone off.
 * Also switch there to reschedule if the reschedule will do something.
 * This happens when
 *      (1) quantum has expired
 *      (2) current process received full quantum (as clock sampled it!)
 *      (3) something else is ready to run.
 * Also call TTY and PRINTER and let them do whatever is necessary.
 *
 * Many global global and static variables are accessed here.  The safety
 * of this must be justified.  Most of them are not changed here:
 *      k_reenter:
 *              This safely tells if the clock interrupt is nested.
 *      proc_ptr, bill_ptr:
 *              These are used for accounting.  It does not matter if proc.c
 *              is changing them, provided they are always valid pointers,
 *              since at worst the previous process would be billed.
 *      next_alarm, realtime, sched_ticks, bill_ptr, prev_ptr,
 *      rdy_head[USER_Q]:
 *              These are tested to decide whether to call interrupt().  It
 *              does not matter if the test is sometimes (rarely) backwards
 *              due to a race, since this will only delay the high-level
 *              processing by one tick, or call the high level unnecessarily.
 * The variables which are changed require more care:
 *      rp->user_time, rp->sys_time:
```

```
 *              These are protected by explicit locks in system.c.  They are
 *              not properly protected in dmp.c (the increment here is not
 *              atomic) but that hardly matters.
 *      pending_ticks:
 *              This is protected by explicit locks in clock.c.  Don't
 *              update realtime directly, since there are too many
 *              references to it to guard conveniently.
 *      lost_ticks:
 *              Clock ticks counted outside the clock task.
 *      sched_ticks, prev_ptr:
 *              Updating these competes with similar code in do_clocktick().
 *              No lock is necessary, because if bad things happen here
 *              (like sched_ticks going negative), the code in do_clocktick()
 *              will restore the variables to reasonable values, and an
 *              occasional missed or extra sched() is harmless.
 *
 * Are these complications worth the trouble?  Well, they make the system 15%
 * faster on a 5MHz 8088, and make task debugging much easier since there are
 * no task switches on an inactive system.
 */

  register struct proc *rp;
  register unsigned ticks;
  clock_t now;

  if (ps_mca) {
        /* Acknowledge the PS/2 clock interrupt. */
        out_byte(PORT_B, in_byte(PORT_B) | CLOCK_ACK_BIT);
  }

  /* Update user and system accounting times.
   * First charge the current process for user time.
   * If the current process is not the billable process (usually because it
   * is a task), charge the billable process for system time as well.
   * Thus the unbillable tasks' user time is the billable users' system time.
   */
  if (k_reenter != 0)
        rp = proc_addr(HARDWARE);
  else
        rp = proc_ptr;
  ticks = lost_ticks + 1;
  lost_ticks = 0;
  rp->user_time += ticks;
  if (rp != bill_ptr && rp != proc_addr(IDLE)) bill_ptr->sys_time += ticks;

  pending_ticks += ticks;
  now = realtime + pending_ticks;
  if (tty_timeout <= now) tty_wakeup(now);      /* possibly wake up TTY */
  pr_restart();                                 /* possibly restart printer */

  if (next_alarm <= now ||
      sched_ticks == 1 &&
      bill_ptr == prev_ptr &&
      rdy_head[USER_Q] != NIL_PROC) {
        interrupt(CLOCK);
        return 1;       /* Reenable interrupts */
  }

  if (--sched_ticks == 0) {
        /* If bill_ptr == prev_ptr, no ready users so don't need sched(). */
```

```
          sched_ticks = SCHED_RATE;       /* reset quantum */
          prev_ptr = bill_ptr;            /* new previous process */
    }
    return 1;      /* Reenable clock interrupt */
  }

  /*===========================================================================*
   *                              init_clock                                   *
   *===========================================================================*/
  PRIVATE void init_clock()
  {
  /* Initialize channel 0 of the 8253A timer to e.g. 60 Hz. */

    out_byte(TIMER_MODE, SQUARE_WAVE);    /* set timer to run continuously */
    out_byte(TIMER0, TIMER_COUNT);        /* load timer low byte */
    out_byte(TIMER0, TIMER_COUNT >> 8);   /* load timer high byte */
    put_irq_handler(CLOCK_IRQ, clock_handler);    /* set the interrupt handler */
    enable_irq(CLOCK_IRQ);                /* ready for clock interrupts */
  }


  /*===========================================================================*
   *                              clock_stop                                   *
   *===========================================================================*/
  PUBLIC void clock_stop()
  {
  /* Reset the clock to the BIOS rate. (For rebooting) */

    out_byte(TIMER_MODE, 0x36);
    out_byte(TIMER0, 0);
    out_byte(TIMER0, 0);
  }


  /*==========================================================================*
   *                              milli_delay                                 *
   *==========================================================================*/
  PUBLIC void milli_delay(millisec)
  unsigned millisec;
  {
  /* Delay some milliseconds. */

    struct milli_state ms;

    milli_start(&ms);
    while (milli_elapsed(&ms) < millisec) {}
  }

  /*==========================================================================*
   *                              milli_start                                 *
   *==========================================================================*/
  PUBLIC void milli_start(msp)
  struct milli_state *msp;
  {
  /* Prepare for calls to milli_elapsed(). */

    msp->prev_count = 0;
    msp->accum_count = 0;
  }
```

```
/*===========================================================================*
 *                              milli_elapsed                                *
 *===========================================================================*/
PUBLIC unsigned milli_elapsed(msp)
struct milli_state *msp;
{
/* Return the number of milliseconds since the call to milli_start().  Must be
 * polled rapidly.
 */
  unsigned count;

  /* Read the counter for channel 0 of the 8253A timer.  The counter
   * decrements at twice the timer frequency (one full cycle for each
   * half of square wave).  The counter normally has a value between 0
   * and TIMER_COUNT, but before the clock task has been initialized,
   * its maximum value is 65535, as set by the BIOS.
   */
  out_byte(TIMER_MODE, LATCH_COUNT);    /* make chip copy count to latch */
  count = in_byte(TIMER0);      /* countdown continues during 2-step read */
  count |= in_byte(TIMER0) << 8;

  /* Add difference between previous and new count unless the counter has
   * increased (restarted its cycle).  We may lose a tick now and then, but
   * microsecond precision is not needed.
   */
  msp->accum_count += count <= msp->prev_count ? (msp->prev_count - count) : 1;
  msp->prev_count = count;

  return msp->accum_count / (TIMER_FREQ / 1000);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/tty.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/*      tty.h - Terminals       */

#define TTY_IN_BYTES     256    /* tty input queue size */
#define TAB_SIZE           8    /* distance between tab stops */
#define TAB_MASK           7    /* mask to compute a tab stop position */

#define ESC             '\33'   /* escape */

#define O_NOCTTY       00400    /* from <fcntl.h>, or cc will choke */
#define O_NONBLOCK     04000

typedef _PROTOTYPE( void (*devfun_t), (struct tty *tp) );
typedef _PROTOTYPE( void (*devfunarg_t), (struct tty *tp, int c) );

typedef struct tty {
  int tty_events;               /* set when TTY should inspect this line */

  /* Input queue.  Typed characters are stored here until read by a program. */
  u16_t *tty_inhead;            /* pointer to place where next char goes */
  u16_t *tty_intail;            /* pointer to next char to be given to prog */
```

```
  int tty_incount;              /* # chars in the input queue */
  int tty_eotct;                /* number of "line breaks" in input queue */
  devfun_t tty_devread;         /* routine to read from low level buffers */
  devfun_t tty_icancel;         /* cancel any device input */
  int tty_min;                  /* minimum requested #chars in input queue */
  clock_t tty_time;             /* time when the input is available */
  struct tty *tty_timenext;     /* for a list of ttys with active timers */

  /* Output section. */
  devfun_t tty_devwrite;        /* routine to start actual device output */
  devfunarg_t tty_echo;         /* routine to echo characters input */
  devfun_t tty_ocancel;         /* cancel any ongoing device output */
  devfun_t tty_break;           /* let the device send a break */

  /* Terminal parameters and status. */
  int tty_position;             /* current position on the screen for echoing */
  char tty_reprint;             /* 1 when echoed input messed up, else 0 */
  char tty_escaped;             /* 1 when LNEXT (^V) just seen, else 0 */
  char tty_inhibited;           /* 1 when STOP (^S) just seen (stops output) */
  char tty_pgrp;                /* slot number of controlling process */
  char tty_openct;              /* count of number of opens of this tty */

  /* Information about incomplete I/O requests is stored here. */
  char tty_inrepcode;           /* reply code, TASK_REPLY or REVIVE */
  char tty_incaller;            /* process that made the call (usually FS) */
  char tty_inproc;              /* process that wants to read from tty */
  vir_bytes tty_in_vir;         /* virtual address where data is to go */
  int tty_inleft;               /* how many chars are still needed */
  int tty_incum;                /* # chars input so far */
  char tty_outrepcode;          /* reply code, TASK_REPLY or REVIVE */
  char tty_outcaller;           /* process that made the call (usually FS) */
  char tty_outproc;             /* process that wants to write to tty */
  vir_bytes tty_out_vir;        /* virtual address where data comes from */
  int tty_outleft;              /* # chars yet to be output */
  int tty_outcum;               /* # chars output so far */
  char tty_iocaller;            /* process that made the call (usually FS) */
  char tty_ioproc;              /* process that wants to do an ioctl */
  int tty_ioreq;                /* ioctl request code */
  vir_bytes tty_iovir;          /* virtual address of ioctl buffer */

  /* Miscellaneous. */
  devfun_t tty_ioctl;           /* set line speed, etc. at the device level */
  devfun_t tty_close;           /* tell the device that the tty is closed */
  void *tty_priv;               /* pointer to per device private data */
  struct termios tty_termios;   /* terminal attributes */
  struct winsize tty_winsize;   /* window size (#lines and #columns) */

  u16_t tty_inbuf[TTY_IN_BYTES];/* tty input buffer */
} tty_t;

EXTERN tty_t tty_table[NR_CONS+NR_RS_LINES+NR_PTYS];

/* Values for the fields. */
#define NOT_ESCAPED        0    /* previous character is not LNEXT (^V) */
#define ESCAPED            1    /* previous character was LNEXT (^V) */
#define RUNNING            0    /* no STOP (^S) has been typed to stop output */
#define STOPPED            1    /* STOP (^S) has been typed to stop output */

/* Fields and flags on characters in the input queue. */
#define IN_CHAR       0x00FF    /* low 8 bits are the character itself */
```

```
#define IN_LEN        0x0F00    /* length of char if it has been echoed */
#define IN_LSHIFT          8    /* length = (c & IN_LEN) >> IN_LSHIFT */
#define IN_EOT        0x1000    /* char is a line break (^D, LF) */
#define IN_EOF        0x2000    /* char is EOF (^D), do not return to user */
#define IN_ESC        0x4000    /* escaped by LNEXT (^V), no interpretation */

/* Times and timeouts. */
#define TIME_NEVER      ((clock_t) -1 < 0 ? (clock_t) LONG_MAX : (clock_t) -1)
#define force_timeout() ((void) (tty_timeout = 0))

EXTERN tty_t *tty_timelist;     /* list of ttys with active timers */

/* Number of elements and limit of a buffer. */
#define buflen(buf)     (sizeof(buf) / sizeof((buf)[0]))
#define bufend(buf)     ((buf) + buflen(buf))
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/tty.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the terminal driver, both for the IBM console and regular
 * ASCII terminals.  It handles only the device-independent part of a TTY, the
 * device dependent parts are in console.c, rs232.c, etc.  This file contains
 * two main entry points, tty_task() and tty_wakeup(), and several minor entry
 * points for use by the device-dependent code.
 *
 * The device-independent part accepts "keyboard" input from the device-
 * dependent part, performs input processing (special key interpretation),
 * and sends the input to a process reading from the TTY.  Output to a TTY
 * is sent to the device-dependent code for output processing and "screen"
 * display.  Input processing is done by the device by calling 'in_process'
 * on the input characters, output processing may be done by the device itself
 * or by calling 'out_process'.  The TTY takes care of input queuing, the
 * device does the output queuing.  If a device receives an external signal,
 * like an interrupt, then it causes tty_wakeup() to be run by the CLOCK task
 * to, you guessed it, wake up the TTY to check if input or output can
 * continue.
 *
 * The valid messages and their parameters are:
 *
 *   HARD_INT:     output has been completed or input has arrived
 *   DEV_READ:     a process wants to read from a terminal
 *   DEV_WRITE:    a process wants to write on a terminal
 *   DEV_IOCTL:    a process wants to change a terminal's parameters
 *   DEV_OPEN:     a tty line has been opened
 *   DEV_CLOSE:    a tty line has been closed
 *   CANCEL:       terminate a previous incomplete system call immediately
 *
 *    m_type      TTY_LINE   PROC_NR    COUNT   TTY_SPEK  TTY_FLAGS  ADDRESS
 * ---------------------------------------------------------------------------
 * | HARD_INT    |         |         |         |         |         |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_READ    |minor dev| proc nr |  count  |         O_NONBLOCK| buf ptr |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_WRITE   |minor dev| proc nr |  count  |         |         | buf ptr |
```

```
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_IOCTL   |minor dev| proc nr |func code|erase etc|  flags  |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_OPEN    |minor dev| proc nr | O_NOCTTY|         |         |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | DEV_CLOSE   |minor dev| proc nr |         |         |         |         |
 * |-------------+---------+---------+---------+---------+---------+---------|
 * | CANCEL      |minor dev| proc nr |         |         |         |         |
 * ---------------------------------------------------------------------------
 */

#include "kernel.h"
#include <termios.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/keymap.h>
#include "tty.h"
#include "proc.h"

/* Address of a tty structure. */
#define tty_addr(line)  (&tty_table[line])

/* First minor numbers for the various classes of TTY devices. */
#define CONS_MINOR        0
#define LOG_MINOR        15
#define RS232_MINOR      16
#define TTYPX_MINOR     128
#define PTYPX_MINOR     192

/* Macros for magic tty types. */
#define isconsole(tp)   ((tp) < tty_addr(NR_CONS))

/* Macros for magic tty structure pointers. */
#define FIRST_TTY       tty_addr(0)
#define END_TTY         tty_addr(sizeof(tty_table) / sizeof(tty_table[0]))

/* A device exists if at least its 'devread' function is defined. */
#define tty_active(tp)  ((tp)->tty_devread != NULL)

/* RS232 lines or pseudo terminals can be completely configured out. */
#if NR_RS_LINES == 0
#define rs_init(tp)     ((void) 0)
#endif
#if NR_PTYS == 0
#define pty_init(tp)    ((void) 0)
#define do_pty(tp, mp)  ((void) 0)
#endif

FORWARD _PROTOTYPE( void do_cancel, (tty_t *tp, message *m_ptr)         );
FORWARD _PROTOTYPE( void do_ioctl, (tty_t *tp, message *m_ptr)          );
FORWARD _PROTOTYPE( void do_open, (tty_t *tp, message *m_ptr)           );
FORWARD _PROTOTYPE( void do_close, (tty_t *tp, message *m_ptr)          );
FORWARD _PROTOTYPE( void do_read, (tty_t *tp, message *m_ptr)           );
FORWARD _PROTOTYPE( void do_write, (tty_t *tp, message *m_ptr)          );
FORWARD _PROTOTYPE( void in_transfer, (tty_t *tp)                       );
FORWARD _PROTOTYPE( int echo, (tty_t *tp, int ch)                       );
FORWARD _PROTOTYPE( void rawecho, (tty_t *tp, int ch)                   );
FORWARD _PROTOTYPE( int back_over, (tty_t *tp)                          );
```

```c
FORWARD _PROTOTYPE( void reprint, (tty_t *tp)                           );
FORWARD _PROTOTYPE( void dev_ioctl, (tty_t *tp)                         );
FORWARD _PROTOTYPE( void setattr, (tty_t *tp)                           );
FORWARD _PROTOTYPE( void tty_icancel, (tty_t *tp)                       );
FORWARD _PROTOTYPE( void tty_init, (tty_t *tp)                          );
FORWARD _PROTOTYPE( void settimer, (tty_t *tp, int on)                  );

/* Default attributes. */
PRIVATE struct termios termios_defaults = {
  TINPUT_DEF, TOUTPUT_DEF, TCTRL_DEF, TLOCAL_DEF, TSPEED_DEF, TSPEED_DEF,
  {
        TEOF_DEF, TEOL_DEF, TERASE_DEF, TINTR_DEF, TKILL_DEF, TMIN_DEF,
        TQUIT_DEF, TTIME_DEF, TSUSP_DEF, TSTART_DEF, TSTOP_DEF,
        TREPRINT_DEF, TLNEXT_DEF, TDISCARD_DEF,
  },
};
PRIVATE struct winsize winsize_defaults;        /* = all zeroes */


/*===========================================================================*
 *                              tty_task                                     *
 *===========================================================================*/
PUBLIC void tty_task()
{
/* Main routine of the terminal task. */

  message tty_mess;             /* buffer for all incoming messages */
  register tty_t *tp;
  unsigned line;

  /* Initialize the terminal lines. */
  for (tp = FIRST_TTY; tp < END_TTY; tp++) tty_init(tp);

  /* Display the Minix startup banner. */
  printf("Minix %s.%s  Copyright 1997 Prentice-Hall, Inc.\n\n",
                                                OS_RELEASE, OS_VERSION);
  printf("Executing in 32-bit protected mode\n\n");

  while (TRUE) {
        /* Handle any events on any of the ttys. */
        for (tp = FIRST_TTY; tp < END_TTY; tp++) {
                if (tp->tty_events) handle_events(tp);
        }

        receive(ANY, &tty_mess);

        /* A hardware interrupt is an invitation to check for events. */
        if (tty_mess.m_type == HARD_INT) continue;

        /* Check the minor device number. */
        line = tty_mess.TTY_LINE;
        if ((line - CONS_MINOR) < NR_CONS) {
                tp = tty_addr(line - CONS_MINOR);
        } else
        if (line == LOG_MINOR) {
                tp = tty_addr(0);
        } else
        if ((line - RS232_MINOR) < NR_RS_LINES) {
                tp = tty_addr(line - RS232_MINOR + NR_CONS);
        } else
```

```
        if ((line - TTYPX_MINOR) < NR_PTYS) {
                tp = tty_addr(line - TTYPX_MINOR + NR_CONS + NR_RS_LINES);
        } else
        if ((line - PTYPX_MINOR) < NR_PTYS) {
                tp = tty_addr(line - PTYPX_MINOR + NR_CONS + NR_RS_LINES);
                do_pty(tp, &tty_mess);
                continue;                       /* this is a pty, not a tty */
        } else {
                tp = NULL;
        }

        /* If the device doesn't exist or is not configured return ENXIO. */
        if (tp == NULL || !tty_active(tp)) {
                tty_reply(TASK_REPLY, tty_mess.m_source,
                                                tty_mess.PROC_NR, ENXIO);
                continue;
        }

        /* Execute the requested function. */
        switch (tty_mess.m_type) {
            case DEV_READ:      do_read(tp, &tty_mess);         break;
            case DEV_WRITE:     do_write(tp, &tty_mess);        break;
            case DEV_IOCTL:     do_ioctl(tp, &tty_mess);        break;
            case DEV_OPEN:      do_open(tp, &tty_mess);         break;
            case DEV_CLOSE:     do_close(tp, &tty_mess);        break;
            case CANCEL:        do_cancel(tp, &tty_mess);       break;
            default:            tty_reply(TASK_REPLY, tty_mess.m_source,
                                                tty_mess.PROC_NR, EINVAL);
        }
  }
}


/*===========================================================================*
 *                              do_read                                      *
 *===========================================================================*/
PRIVATE void do_read(tp, m_ptr)
register tty_t *tp;             /* pointer to tty struct */
message *m_ptr;                 /* pointer to message sent to the task */
{
/* A process wants to read from a terminal. */
  int r;

  /* Check if there is already a process hanging in a read, check if the
   * parameters are correct, do I/O.
   */
  if (tp->tty_inleft > 0) {
        r = EIO;
  } else
  if (m_ptr->COUNT <= 0) {
        r = EINVAL;
  } else
  if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
        r = EFAULT;
  } else {
        /* Copy information from the message to the tty struct. */
        tp->tty_inrepcode = TASK_REPLY;
        tp->tty_incaller = m_ptr->m_source;
        tp->tty_inproc = m_ptr->PROC_NR;
        tp->tty_in_vir = (vir_bytes) m_ptr->ADDRESS;
```

```
        tp->tty_inleft = m_ptr->COUNT;

        if (!(tp->tty_termios.c_lflag & ICANON)
                                        && tp->tty_termios.c_cc[VTIME] > 0) {
                if (tp->tty_termios.c_cc[VMIN] == 0) {
                        /* MIN & TIME specify a read timer that finishes the
                         * read in TIME/10 seconds if no bytes are available.
                         */
                        lock();
                        settimer(tp, TRUE);
                        tp->tty_min = 1;
                        unlock();
                } else {
                        /* MIN & TIME specify an inter-byte timer that may
                         * have to be cancelled if there are no bytes yet.
                         */
                        if (tp->tty_eotct == 0) {
                                lock();
                                settimer(tp, FALSE);
                                unlock();
                                tp->tty_min = tp->tty_termios.c_cc[VMIN];
                        }
                }
        }

        /* Anything waiting in the input buffer? Clear it out... */
        in_transfer(tp);
        /* ...then go back for more */
        handle_events(tp);
        if (tp->tty_inleft == 0) return;                /* already done */

        /* There were no bytes in the input queue available, so either suspend
         * the caller or break off the read if nonblocking.
         */
        if (m_ptr->TTY_FLAGS & O_NONBLOCK) {
                r = EAGAIN;                             /* cancel the read */
                tp->tty_inleft = tp->tty_incum = 0;
        } else {
                r = SUSPEND;                            /* suspend the caller */
                tp->tty_inrepcode = REVIVE;
        }
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}


/*==========================================================================*
 *                              do_write                                    *
 *==========================================================================*/
PRIVATE void do_write(tp, m_ptr)
register tty_t *tp;
register message *m_ptr;        /* pointer to message sent to the task */
{
/* A process wants to write on a terminal. */
  int r;

  /* Check if there is already a process hanging in a write, check if the
   * parameters are correct, do I/O.
   */
  if (tp->tty_outleft > 0) {
```

```
        r = EIO;
  } else
  if (m_ptr->COUNT <= 0) {
        r = EINVAL;
  } else
  if (numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, m_ptr->COUNT) == 0) {
        r = EFAULT;
  } else {
        /* Copy message parameters to the tty structure. */
        tp->tty_outrepcode = TASK_REPLY;
        tp->tty_outcaller = m_ptr->m_source;
        tp->tty_outproc = m_ptr->PROC_NR;
        tp->tty_out_vir = (vir_bytes) m_ptr->ADDRESS;
        tp->tty_outleft = m_ptr->COUNT;

        /* Try to write. */
        handle_events(tp);
        if (tp->tty_outleft == 0) return;               /* already done */

        /* None or not all the bytes could be written, so either suspend the
         * caller or break off the write if nonblocking.
         */
        if (m_ptr->TTY_FLAGS & O_NONBLOCK) {            /* cancel the write */
                r = tp->tty_outcum > 0 ? tp->tty_outcum : EAGAIN;
                tp->tty_outleft = tp->tty_outcum = 0;
        } else {
                r = SUSPEND;                            /* suspend the caller */
                tp->tty_outrepcode = REVIVE;
        }
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}


/*===========================================================================*
 *                              do_ioctl                                     *
 *===========================================================================*/
PRIVATE void do_ioctl(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                 /* pointer to message sent to task */
{
/* Perform an IOCTL on this terminal. Posix termios calls are handled
 * by the IOCTL system call
 */

  int r;
  union {
        int i;
        /* these non-Posix params are not used now, but the union is retained
         * to minimize code differences with backward compatible version
         * struct sgttyb sg;
         * struct tchars tc;
         */
  } param;
  phys_bytes user_phys;
  size_t size;

  /* Size of the ioctl parameter. */
  switch (m_ptr->TTY_REQUEST) {
    case TCGETS:        /* Posix tcgetattr function */
```

```
          case TCSETS:        /* Posix tcsetattr function, TCSANOW option */
          case TCSETSW:       /* Posix tcsetattr function, TCSADRAIN option */
          case TCSETSF:       /* Posix tcsetattr function, TCSAFLUSH option */
              size = sizeof(struct termios);
              break;

          case TCSBRK:        /* Posix tcsendbreak function */
          case TCFLOW:        /* Posix tcflow function */
          case TCFLSH:        /* Posix tcflush function */
          case TIOCGPGRP:     /* Posix tcgetpgrp function */
          case TIOCSPGRP:     /* Posix tcsetpgrp function */
              size = sizeof(int);
              break;

          case TIOCGWINSZ:    /* get window size (not Posix) */
          case TIOCSWINSZ:    /* set window size (not Posix) */
              size = sizeof(struct winsize);
              break;

          case KIOCSMAP:      /* load keymap (Minix extension) */
              size = sizeof(keymap_t);
              break;

          case TIOCSFON:      /* load font (Minix extension) */
              size = sizeof(u8_t [8192]);
              break;

          case TCDRAIN:       /* Posix tcdrain function -- no parameter */
          default:            size = 0;
        }

        if (size != 0) {
              user_phys = numap(m_ptr->PROC_NR, (vir_bytes) m_ptr->ADDRESS, size);
              if (user_phys == 0) {
                      tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, EFAULT);
                      return;
              }
        }

        r = OK;
        switch (m_ptr->TTY_REQUEST) {
          case TCGETS:
              /* Get the termios attributes. */
              phys_copy(vir2phys(&tp->tty_termios), user_phys, (phys_bytes) size);
              break;

          case TCSETSW:
          case TCSETSF:
          case TCDRAIN:
              if (tp->tty_outleft > 0) {
                      /* Wait for all ongoing output processing to finish. */
                      tp->tty_iocaller = m_ptr->m_source;
                      tp->tty_ioproc = m_ptr->PROC_NR;
                      tp->tty_ioreq = m_ptr->REQUEST;
                      tp->tty_iovir = (vir_bytes) m_ptr->ADDRESS;
                      r = SUSPEND;
                      break;
              }
              if (m_ptr->TTY_REQUEST == TCDRAIN) break;
              if (m_ptr->TTY_REQUEST == TCSETSF) tty_icancel(tp);
```

```
            /*FALL THROUGH*/
        case TCSETS:
            /* Set the termios attributes. */
            phys_copy(user_phys, vir2phys(&tp->tty_termios), (phys_bytes) size);
            setattr(tp);
            break;

        case TCFLSH:
            phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
            switch (param.i) {
                case TCIFLUSH:      tty_icancel(tp);                        break;
                case TCOFLUSH:      (*tp->tty_ocancel)(tp);                 break;
                case TCIOFLUSH:     tty_icancel(tp); (*tp->tty_ocancel)(tp);break;
                default:            r = EINVAL;
            }
            break;

        case TCFLOW:
            phys_copy(user_phys, vir2phys(&param.i), (phys_bytes) size);
            switch (param.i) {
                case TCOOFF:
                case TCOON:
                    tp->tty_inhibited = (param.i == TCOOFF);
                    tp->tty_events = 1;
                    break;
                case TCIOFF:
                    (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTOP]);
                    break;
                case TCION:
                    (*tp->tty_echo)(tp, tp->tty_termios.c_cc[VSTART]);
                    break;
                default:
                    r = EINVAL;
            }
            break;

        case TCSBRK:
            if (tp->tty_break != NULL) (*tp->tty_break)(tp);
            break;

        case TIOCGWINSZ:
            phys_copy(vir2phys(&tp->tty_winsize), user_phys, (phys_bytes) size);
            break;

        case TIOCSWINSZ:
            phys_copy(user_phys, vir2phys(&tp->tty_winsize), (phys_bytes) size);
            /* SIGWINCH... */
            break;

        case KIOCSMAP:
            /* Load a new keymap (only /dev/console). */
            if (isconsole(tp)) r = kbd_loadmap(user_phys);
            break;

        case TIOCSFON:
            /* Load a font into an EGA or VGA card (hs@hck.hr) */
            if (isconsole(tp)) r = con_loadfont(user_phys);
            break;

    /* These Posix functions are allowed to fail if _POSIX_JOB_CONTROL is
```

```
 * not defined.
 */
    case TIOCGPGRP:
    case TIOCSPGRP:
    default:
        r = ENOTTY;
  }

  /* Send the reply. */
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}


/*===========================================================================*
 *                              do_open                                      *
 *===========================================================================*/
PRIVATE void do_open(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                 /* pointer to message sent to task */
{
/* A tty line has been opened.  Make it the callers controlling tty if
 * O_NOCTTY is *not* set and it is not the log device.  1 is returned if
 * the tty is made the controlling tty, otherwise OK or an error code.
 */
  int r = OK;

  if (m_ptr->TTY_LINE == LOG_MINOR) {
        /* The log device is a write-only diagnostics device. */
        if (m_ptr->COUNT & R_BIT) r = EACCES;
  } else {
        if (!(m_ptr->COUNT & O_NOCTTY)) {
                tp->tty_pgrp = m_ptr->PROC_NR;
                r = 1;
        }
        tp->tty_openct++;
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, r);
}


/*===========================================================================*
 *                              do_close                                     *
 *===========================================================================*/
PRIVATE void do_close(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                 /* pointer to message sent to task */
{
/* A tty line has been closed.  Clean up the line if it is the last close. */

  if (m_ptr->TTY_LINE != LOG_MINOR && --tp->tty_openct == 0) {
        tp->tty_pgrp = 0;
        tty_icancel(tp);
        (*tp->tty_ocancel)(tp);
        (*tp->tty_close)(tp);
        tp->tty_termios = termios_defaults;
        tp->tty_winsize = winsize_defaults;
        setattr(tp);
  }
  tty_reply(TASK_REPLY, m_ptr->m_source, m_ptr->PROC_NR, OK);
}
```

```
/*==========================================================================*
 *                              do_cancel                                   *
 *==========================================================================*/
PRIVATE void do_cancel(tp, m_ptr)
register tty_t *tp;
message *m_ptr;                 /* pointer to message sent to task */
{
/* A signal has been sent to a process that is hanging trying to read or write.
 * The pending read or write must be finished off immediately.
 */

  int proc_nr;
  int mode;

  /* Check the parameters carefully, to avoid cancelling twice. */
  proc_nr = m_ptr->PROC_NR;
  mode = m_ptr->COUNT;
  if ((mode & R_BIT) && tp->tty_inleft != 0 && proc_nr == tp->tty_inproc) {
        /* Process was reading when killed.  Clean up input. */
        tty_icancel(tp);
        tp->tty_inleft = tp->tty_incum = 0;
  }
  if ((mode & W_BIT) && tp->tty_outleft != 0 && proc_nr == tp->tty_outproc) {
        /* Process was writing when killed.  Clean up output. */
        (*tp->tty_ocancel)(tp);
        tp->tty_outleft = tp->tty_outcum = 0;
  }
  if (tp->tty_ioreq != 0 && proc_nr == tp->tty_ioproc) {
        /* Process was waiting for output to drain. */
        tp->tty_ioreq = 0;
  }
  tp->tty_events = 1;
  tty_reply(TASK_REPLY, m_ptr->m_source, proc_nr, EINTR);
}


/*==========================================================================*
 *                              handle_events                               *
 *==========================================================================*/
PUBLIC void handle_events(tp)
tty_t *tp;                      /* TTY to check for events. */
{
/* Handle any events pending on a TTY.  These events are usually device
 * interrupts.
 *
 * Two kinds of events are prominent:
 *      - a character has been received from the console or an RS232 line.
 *      - an RS232 line has completed a write request (on behalf of a user).
 * The interrupt handler may delay the interrupt message at its discretion
 * to avoid swamping the TTY task.  Messages may be overwritten when the
 * lines are fast or when there are races between different lines, input
 * and output, because MINIX only provides single buffering for interrupt
 * messages (in proc.c).  This is handled by explicitly checking each line
 * for fresh input and completed output on each interrupt.
 */
  char *buf;
  unsigned count;

```

```
  do {
        tp->tty_events = 0;

        /* Read input and perform input processing. */
        (*tp->tty_devread)(tp);

        /* Perform output processing and write output. */
        (*tp->tty_devwrite)(tp);

        /* Ioctl waiting for some event? */
        if (tp->tty_ioreq != 0) dev_ioctl(tp);
  } while (tp->tty_events);

  /* Transfer characters from the input queue to a waiting process. */
  in_transfer(tp);

  /* Reply if enough bytes are available. */
  if (tp->tty_incum >= tp->tty_min && tp->tty_inleft > 0) {
        tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
                                                        tp->tty_incum);
        tp->tty_inleft = tp->tty_incum = 0;
  }
}


/*==========================================================================*
 *                              in_transfer                                 *
 *==========================================================================*/
PRIVATE void in_transfer(tp)
register tty_t *tp;             /* pointer to terminal to read from */
{
/* Transfer bytes from the input queue to a process reading from a terminal. */

  int ch;
  int count;
  phys_bytes buf_phys, user_base;
  char buf[64], *bp;

  /* Anything to do? */
  if (tp->tty_inleft == 0 || tp->tty_eotct < tp->tty_min) return;

  buf_phys = vir2phys(buf);
  user_base = proc_vir2phys(proc_addr(tp->tty_inproc), 0);
  bp = buf;
  while (tp->tty_inleft > 0 && tp->tty_eotct > 0) {
        ch = *tp->tty_intail;

        if (!(ch & IN_EOF)) {
                /* One character to be delivered to the user. */
                *bp = ch & IN_CHAR;
                tp->tty_inleft--;
                if (++bp == bufend(buf)) {
                        /* Temp buffer full, copy to user space. */
                        phys_copy(buf_phys, user_base + tp->tty_in_vir,
                                                (phys_bytes) buflen(buf));
                        tp->tty_in_vir += buflen(buf);
                        tp->tty_incum += buflen(buf);
                        bp = buf;
                }
        }
```

```

          /* Remove the character from the input queue. */
          if (++tp->tty_intail == bufend(tp->tty_inbuf))
                  tp->tty_intail = tp->tty_inbuf;
          tp->tty_incount--;
          if (ch & IN_EOT) {
                  tp->tty_eotct--;
                  /* Don't read past a line break in canonical mode. */
                  if (tp->tty_termios.c_lflag & ICANON) tp->tty_inleft = 0;
          }
  }

  if (bp > buf) {
          /* Leftover characters in the buffer. */
          count = bp - buf;
          phys_copy(buf_phys, user_base + tp->tty_in_vir, (phys_bytes) count);
          tp->tty_in_vir += count;
          tp->tty_incum += count;
  }

  /* Usually reply to the reader, possibly even if incum == 0 (EOF). */
  if (tp->tty_inleft == 0) {
          tty_reply(tp->tty_inrepcode, tp->tty_incaller, tp->tty_inproc,
                                                                tp->tty_incum);
          tp->tty_inleft = tp->tty_incum = 0;
  }
}


/*==========================================================================*
 *                              in_process                                  *
 *==========================================================================*/
PUBLIC int in_process(tp, buf, count)
register tty_t *tp;             /* terminal on which character has arrived */
char *buf;                      /* buffer with input characters */
int count;                      /* number of input characters */
{
/* Characters have just been typed in.  Process, save, and echo them.  Return
 * the number of characters processed.
 */

  int ch, sig, ct;
  int timeset = FALSE;
  static unsigned char csize_mask[] = { 0x1F, 0x3F, 0x7F, 0xFF };

  for (ct = 0; ct < count; ct++) {
          /* Take one character. */
          ch = *buf++ & BYTE;

          /* Strip to seven bits? */
          if (tp->tty_termios.c_iflag & ISTRIP) ch &= 0x7F;

          /* Input extensions? */
          if (tp->tty_termios.c_lflag & IEXTEN) {

                  /* Previous character was a character escape? */
                  if (tp->tty_escaped) {
                          tp->tty_escaped = NOT_ESCAPED;
                          ch |= IN_ESC;   /* protect character */
                  }
```

```

		/* LNEXT (^V) to escape the next character? */
		if (ch == tp->tty_termios.c_cc[VLNEXT]) {
			tp->tty_escaped = ESCAPED;
			rawecho(tp, '^');
			rawecho(tp, '\b');
			continue;	/* do not store the escape */
		}

		/* REPRINT (^R) to reprint echoed characters? */
		if (ch == tp->tty_termios.c_cc[VREPRINT]) {
			reprint(tp);
			continue;
		}
	}

	/* _POSIX_VDISABLE is a normal character value, so better escape it. */
	if (ch == _POSIX_VDISABLE) ch |= IN_ESC;

	/* Map CR to LF, ignore CR, or map LF to CR. */
	if (ch == '\r') {
		if (tp->tty_termios.c_iflag & IGNCR) continue;
		if (tp->tty_termios.c_iflag & ICRNL) ch = '\n';
	} else
	if (ch == '\n') {
		if (tp->tty_termios.c_iflag & INLCR) ch = '\r';
	}

	/* Canonical mode? */
	if (tp->tty_termios.c_lflag & ICANON) {

		/* Erase processing (rub out of last character). */
		if (ch == tp->tty_termios.c_cc[VERASE]) {
			(void) back_over(tp);
			if (!(tp->tty_termios.c_lflag & ECHOE)) {
				(void) echo(tp, ch);
			}
			continue;
		}

		/* Kill processing (remove current line). */
		if (ch == tp->tty_termios.c_cc[VKILL]) {
			while (back_over(tp)) {}
			if (!(tp->tty_termios.c_lflag & ECHOE)) {
				(void) echo(tp, ch);
				if (tp->tty_termios.c_lflag & ECHOK)
					rawecho(tp, '\n');
			}
			continue;
		}

		/* EOF (^D) means end-of-file, an invisible "line break". */
		if (ch == tp->tty_termios.c_cc[VEOF]) ch |= IN_EOT | IN_EOF;

		/* The line may be returned to the user after an LF. */
		if (ch == '\n') ch |= IN_EOT;

		/* Same thing with EOL, whatever it may be. */
		if (ch == tp->tty_termios.c_cc[VEOL]) ch |= IN_EOT;
	}
```

```

		/* Start/stop input control? */
		if (tp->tty_termios.c_iflag & IXON) {

			/* Output stops on STOP (^S). */
			if (ch == tp->tty_termios.c_cc[VSTOP]) {
				tp->tty_inhibited = STOPPED;
				tp->tty_events = 1;
				continue;
			}

			/* Output restarts on START (^Q) or any character if IXANY. */
			if (tp->tty_inhibited) {
				if (ch == tp->tty_termios.c_cc[VSTART]
						|| (tp->tty_termios.c_iflag & IXANY)) {
					tp->tty_inhibited = RUNNING;
					tp->tty_events = 1;
					if (ch == tp->tty_termios.c_cc[VSTART])
						continue;
				}
			}
		}

		if (tp->tty_termios.c_lflag & ISIG) {
			/* Check for INTR (^?) and QUIT (^\) characters. */
			if (ch == tp->tty_termios.c_cc[VINTR]
						|| ch == tp->tty_termios.c_cc[VQUIT]) {
				sig = SIGINT;
				if (ch == tp->tty_termios.c_cc[VQUIT]) sig = SIGQUIT;
				sigchar(tp, sig);
				(void) echo(tp, ch);
				continue;
			}
		}

		/* Is there space in the input buffer? */
		if (tp->tty_incount == buflen(tp->tty_inbuf)) {
			/* No space; discard in canonical mode, keep in raw mode. */
			if (tp->tty_termios.c_lflag & ICANON) continue;
			break;
		}

		if (!(tp->tty_termios.c_lflag & ICANON)) {
			/* In raw mode all characters are "line breaks". */
			ch |= IN_EOT;

			/* Start an inter-byte timer? */
			if (!timeset && tp->tty_termios.c_cc[VMIN] > 0
					&& tp->tty_termios.c_cc[VTIME] > 0) {
				lock();
				settimer(tp, TRUE);
				unlock();
				timeset = TRUE;
			}
		}

		/* Perform the intricate function of echoing. */
		if (tp->tty_termios.c_lflag & (ECHO|ECHONL)) ch = echo(tp, ch);

		/* Save the character in the input queue. */
```

```
        *tp->tty_inhead++ = ch;
        if (tp->tty_inhead == bufend(tp->tty_inbuf))
                tp->tty_inhead = tp->tty_inbuf;
        tp->tty_incount++;
        if (ch & IN_EOT) tp->tty_eotct++;

        /* Try to finish input if the queue threatens to overflow. */
        if (tp->tty_incount == buflen(tp->tty_inbuf)) in_transfer(tp);
  }
  return ct;
}


/*==========================================================================*
 *                              echo                                        *
 *==========================================================================*/
PRIVATE int echo(tp, ch)
register tty_t *tp;             /* terminal on which to echo */
register int ch;                /* pointer to character to echo */
{
/* Echo the character if echoing is on.  Some control characters are echoed
 * with their normal effect, other control characters are echoed as "^X",
 * normal characters are echoed normally.  EOF (^D) is echoed, but immediately
 * backspaced over.  Return the character with the echoed length added to its
 * attributes.
 */
  int len, rp;

  ch &= ~IN_LEN;
  if (!(tp->tty_termios.c_lflag & ECHO)) {
        if (ch == ('\n' | IN_EOT) && (tp->tty_termios.c_lflag
                                        & (ICANON|ECHONL)) == (ICANON|ECHONL))
                (*tp->tty_echo)(tp, '\n');
        return(ch);
  }

  /* "Reprint" tells if the echo output has been messed up by other output. */
  rp = tp->tty_incount == 0 ? FALSE : tp->tty_reprint;

  if ((ch & IN_CHAR) < ' ') {
        switch (ch & (IN_ESC|IN_EOF|IN_EOT|IN_CHAR)) {
            case '\t':
                len = 0;
                do {
                        (*tp->tty_echo)(tp, ' ');
                        len++;
                } while (len < TAB_SIZE && (tp->tty_position & TAB_MASK) != 0);
                break;
            case '\r' | IN_EOT:
            case '\n' | IN_EOT:
                (*tp->tty_echo)(tp, ch & IN_CHAR);
                len = 0;
                break;
            default:
                (*tp->tty_echo)(tp, '^');
                (*tp->tty_echo)(tp, '@' + (ch & IN_CHAR));
                len = 2;
        }
  } else
  if ((ch & IN_CHAR) == '\177') {
```

```
                /* A DEL prints as "^?". */
                (*tp->tty_echo)(tp, '^');
                (*tp->tty_echo)(tp, '?');
                len = 2;
        } else {
                (*tp->tty_echo)(tp, ch & IN_CHAR);
                len = 1;
        }
        if (ch & IN_EOF) while (len > 0) { (*tp->tty_echo)(tp, '\b'); len--; }

        tp->tty_reprint = rp;
        return(ch | (len << IN_LSHIFT));
}


/*===========================================================================*
 *                              rawecho                                      *
 *===========================================================================*/
PRIVATE void rawecho(tp, ch)
register tty_t *tp;
int ch;
{
/* Echo without interpretation if ECHO is set. */
  int rp = tp->tty_reprint;
  if (tp->tty_termios.c_lflag & ECHO) (*tp->tty_echo)(tp, ch);
  tp->tty_reprint = rp;
}


/*===========================================================================*
 *                              back_over                                    *
 *===========================================================================*/
PRIVATE int back_over(tp)
register tty_t *tp;
{
/* Backspace to previous character on screen and erase it. */
  u16_t *head;
  int len;

  if (tp->tty_incount == 0) return(0);  /* queue empty */
  head = tp->tty_inhead;
  if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
  if (*--head & IN_EOT) return(0);              /* can't erase "line breaks" */
  if (tp->tty_reprint) reprint(tp);             /* reprint if messed up */
  tp->tty_inhead = head;
  tp->tty_incount--;
  if (tp->tty_termios.c_lflag & ECHOE) {
        len = (*head & IN_LEN) >> IN_LSHIFT;
        while (len > 0) {
                rawecho(tp, '\b');
                rawecho(tp, ' ');
                rawecho(tp, '\b');
                len--;
        }
  }
  return(1);                            /* one character erased */
}


/*===========================================================================*
```

```
 *                               reprint                                  *
 *=========================================================================*/
PRIVATE void reprint(tp)
register tty_t *tp;             /* pointer to tty struct */
{
/* Restore what has been echoed to screen before if the user input has been
 * messed up by output, or if REPRINT (^R) is typed.
 */
  int count;
  u16_t *head;

  tp->tty_reprint = FALSE;

  /* Find the last line break in the input. */
  head = tp->tty_inhead;
  count = tp->tty_incount;
  while (count > 0) {
        if (head == tp->tty_inbuf) head = bufend(tp->tty_inbuf);
        if (head[-1] & IN_EOT) break;
        head--;
        count--;
  }
  if (count == tp->tty_incount) return;         /* no reason to reprint */

  /* Show REPRINT (^R) and move to a new line. */
  (void) echo(tp, tp->tty_termios.c_cc[VREPRINT] | IN_ESC);
  rawecho(tp, '\r');
  rawecho(tp, '\n');

  /* Reprint from the last break onwards. */
  do {
        if (head == bufend(tp->tty_inbuf)) head = tp->tty_inbuf;
        *head = echo(tp, *head);
        head++;
        count++;
  } while (count < tp->tty_incount);
}


/*=========================================================================*
 *                               out_process                              *
 *=========================================================================*/
PUBLIC void out_process(tp, bstart, bpos, bend, icount, ocount)
tty_t *tp;
char *bstart, *bpos, *bend;     /* start/pos/end of circular buffer */
int *icount;                    /* # input chars / input chars used */
int *ocount;                    /* max output chars / output chars used */
{
/* Perform output processing on a circular buffer.  *icount is the number of
 * bytes to process, and the number of bytes actually processed on return.
 * *ocount is the space available on input and the space used on output.
 * (Naturally *icount < *ocount.)  The column position is updated modulo
 * the TAB size, because we really only need it for tabs.
 */

  int tablen;
  int ict = *icount;
  int oct = *ocount;
  int pos = tp->tty_position;

```

```
  while (ict > 0) {
        switch (*bpos) {
        case '\7':
                break;
        case '\b':
                pos--;
                break;
        case '\r':
                pos = 0;
                break;
        case '\n':
                if ((tp->tty_termios.c_oflag & (OPOST|ONLCR))
                                                        == (OPOST|ONLCR)) {
                        /* Map LF to CR+LF if there is space.  Note that the
                         * next character in the buffer is overwritten, so
                         * we stop at this point.
                         */
                        if (oct >= 2) {
                                *bpos = '\r';
                                if (++bpos == bend) bpos = bstart;
                                *bpos = '\n';
                                pos = 0;
                                ict--;
                                oct -= 2;
                        }
                        goto out_done;  /* no space or buffer got changed */
                }
                break;
        case '\t':
                /* Best guess for the tab length. */
                tablen = TAB_SIZE - (pos & TAB_MASK);

                if ((tp->tty_termios.c_oflag & (OPOST|XTABS))
                                                        == (OPOST|XTABS)) {
                        /* Tabs must be expanded. */
                        if (oct >= tablen) {
                                pos += tablen;
                                ict--;
                                oct -= tablen;
                                do {
                                        *bpos = ' ';
                                        if (++bpos == bend) bpos = bstart;
                                } while (--tablen != 0);
                        }
                        goto out_done;
                }
                /* Tabs are output directly. */
                pos += tablen;
                break;
        default:
                /* Assume any other character prints as one character. */
                pos++;
        }
        if (++bpos == bend) bpos = bstart;
        ict--;
        oct--;
  }
out_done:
  tp->tty_position = pos & TAB_MASK;
```

```
  *icount -= ict;       /* [io]ct are the number of chars not used */
  *ocount -= oct;       /* *[io]count are the number of chars that are used */
}


/*==========================================================================*
 *                              dev_ioctl                                   *
 *==========================================================================*/
PRIVATE void dev_ioctl(tp)
tty_t *tp;
{
/* The ioctl's TCSETSW, TCSETSF and TCDRAIN wait for output to finish to make
 * sure that an attribute change doesn't affect the processing of current
 * output.  Once output finishes the ioctl is executed as in do_ioctl().
 */
  phys_bytes user_phys;

  if (tp->tty_outleft > 0) return;              /* output not finished */

  if (tp->tty_ioreq != TCDRAIN) {
        if (tp->tty_ioreq == TCSETSF) tty_icancel(tp);
        user_phys = proc_vir2phys(proc_addr(tp->tty_ioproc), tp->tty_iovir);
        phys_copy(user_phys, vir2phys(&tp->tty_termios),
                                        (phys_bytes) sizeof(tp->tty_termios));
        setattr(tp);
  }
  tp->tty_ioreq = 0;
  tty_reply(REVIVE, tp->tty_iocaller, tp->tty_ioproc, OK);
}


/*==========================================================================*
 *                              setattr                                     *
 *==========================================================================*/
PRIVATE void setattr(tp)
tty_t *tp;
{
/* Apply the new line attributes (raw/canonical, line speed, etc.) */
  u16_t *inp;
  int count;

  if (!(tp->tty_termios.c_lflag & ICANON)) {
        /* Raw mode; put a "line break" on all characters in the input queue.
         * It is undefined what happens to the input queue when ICANON is
         * switched off, a process should use TCSAFLUSH to flush the queue.
         * Keeping the queue to preserve typeahead is the Right Thing, however
         * when a process does use TCSANOW to switch to raw mode.
         */
        count = tp->tty_eotct = tp->tty_incount;
        inp = tp->tty_intail;
        while (count > 0) {
                *inp |= IN_EOT;
                if (++inp == bufend(tp->tty_inbuf)) inp = tp->tty_inbuf;
                --count;
        }
  }

  /* Inspect MIN and TIME. */
  lock();
  settimer(tp, FALSE);
```

```
  unlock();
  if (tp->tty_termios.c_lflag & ICANON) {
	/* No MIN & TIME in canonical mode. */
	tp->tty_min = 1;
  } else {
	/* In raw mode MIN is the number of chars wanted, and TIME how long
	 * to wait for them.  With interesting exceptions if either is zero.
	 */
	tp->tty_min = tp->tty_termios.c_cc[VMIN];
	if (tp->tty_min == 0 && tp->tty_termios.c_cc[VTIME] > 0)
		tp->tty_min = 1;
  }

  if (!(tp->tty_termios.c_iflag & IXON)) {
	/* No start/stop output control, so don't leave output inhibited. */
	tp->tty_inhibited = RUNNING;
	tp->tty_events = 1;
  }

  /* Setting the output speed to zero hangs up the phone. */
  if (tp->tty_termios.c_ospeed == B0) sigchar(tp, SIGHUP);

  /* Set new line speed, character size, etc at the device level. */
  (*tp->tty_ioctl)(tp);
}


/*==========================================================================*
 *				tty_reply				    *
 *==========================================================================*/
PUBLIC void tty_reply(code, replyee, proc_nr, status)
int code;			/* TASK_REPLY or REVIVE */
int replyee;			/* destination address for the reply */
int proc_nr;			/* to whom should the reply go? */
int status;			/* reply code */
{
/* Send a reply to a process that wanted to read or write data. */

  message tty_mess;

  tty_mess.m_type = code;
  tty_mess.REP_PROC_NR = proc_nr;
  tty_mess.REP_STATUS = status;
  if ((status = send(replyee, &tty_mess)) != OK)
	panic("tty_reply failed, status\n", status);
}


/*==========================================================================*
 *				sigchar					    *
 *==========================================================================*/
PUBLIC void sigchar(tp, sig)
register tty_t *tp;
int sig;			/* SIGINT, SIGQUIT, SIGKILL or SIGHUP */
{
/* Process a SIGINT, SIGQUIT or SIGKILL char from the keyboard or SIGHUP from
 * a tty close, "stty 0", or a real RS-232 hangup.  MM will send the signal to
 * the process group (INT, QUIT), all processes (KILL), or the session leader
 * (HUP).
 */
```

```c

  if (tp->tty_pgrp != 0) cause_sig(tp->tty_pgrp, sig);

  if (!(tp->tty_termios.c_lflag & NOFLSH)) {
        tp->tty_incount = tp->tty_eotct = 0;    /* kill earlier input */
        tp->tty_intail = tp->tty_inhead;
        (*tp->tty_ocancel)(tp);                 /* kill all output */
        tp->tty_inhibited = RUNNING;
        tp->tty_events = 1;
  }
}


/*==========================================================================*
 *                              tty_icancel                                 *
 *==========================================================================*/
PRIVATE void tty_icancel(tp)
register tty_t *tp;
{
/* Discard all pending input, tty buffer or device. */

  tp->tty_incount = tp->tty_eotct = 0;
  tp->tty_intail = tp->tty_inhead;
  (*tp->tty_icancel)(tp);
}


/*==========================================================================*
 *                              tty_init                                    *
 *==========================================================================*/
PRIVATE void tty_init(tp)
tty_t *tp;                      /* TTY line to initialize. */
{
/* Initialize tty structure and call device initialization routines. */

  tp->tty_intail = tp->tty_inhead = tp->tty_inbuf;
  tp->tty_min = 1;
  tp->tty_termios = termios_defaults;
  tp->tty_icancel = tp->tty_ocancel = tp->tty_ioctl = tp->tty_close =
                                                                tty_devnop;
  if (tp < tty_addr(NR_CONS)) {
        scr_init(tp);
  } else
  if (tp < tty_addr(NR_CONS+NR_RS_LINES)) {
        rs_init(tp);
  } else {
        pty_init(tp);
  }
}


/*==========================================================================*
 *                              tty_wakeup                                  *
 *==========================================================================*/
PUBLIC void tty_wakeup(now)
clock_t now;                            /* current time */
{
/* Wake up TTY when something interesting is happening on one of the terminal
 * lines, like a character arriving on an RS232 line, a key being typed, or
 * a timer on a line expiring by TIME.
```

```
 */
  tty_t *tp;

  /* Scan the timerlist for expired timers and compute the next timeout time. */
  tty_timeout = TIME_NEVER;
  while ((tp = tty_timelist) != NULL) {
        if (tp->tty_time > now) {
                tty_timeout = tp->tty_time;     /* this timer is next */
                break;
        }
        tp->tty_min = 0;                        /* force read to succeed */
        tp->tty_events = 1;
        tty_timelist = tp->tty_timenext;
  }

  /* Let TTY know there is something afoot. */
  interrupt(TTY);
}


/*==========================================================================*
 *                              settimer                                    *
 *==========================================================================*/
PRIVATE void settimer(tp, on)
tty_t *tp;                      /* line to set or unset a timer on */
int on;                         /* set timer if true, otherwise unset */
{
/* Set or unset a TIME inspired timer.  This function is interrupt sensitive
 * due to tty_wakeup(), so it must be called from within lock()/unlock().
 */
  tty_t **ptp;

  /* Take tp out of the timerlist if present. */
  for (ptp = &tty_timelist; *ptp != NULL; ptp = &(*ptp)->tty_timenext) {
        if (tp == *ptp) {
                *ptp = tp->tty_timenext;        /* take tp out of the list */
                break;
        }
  }
  if (!on) return;                              /* unsetting it is enough */

  /* Timeout occurs TIME deciseconds from now. */
  tp->tty_time = get_uptime() + tp->tty_termios.c_cc[VTIME] * (HZ/10);

  /* Find a new place in the list. */
  for (ptp = &tty_timelist; *ptp != NULL; ptp = &(*ptp)->tty_timenext) {
        if (tp->tty_time <= (*ptp)->tty_time) break;
  }
  tp->tty_timenext = *ptp;
  *ptp = tp;
  if (tp->tty_time < tty_timeout) tty_timeout = tp->tty_time;
}


/*==========================================================================*
 *                              tty_devnop                                  *
 *==========================================================================*/
PUBLIC void tty_devnop(tp)
tty_t *tp;
{
```

```
  /* Some functions need not be implemented at the device level. */
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/keyboard.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Keyboard driver for PC's and AT's.
 *
 * Changed by Marcus Hampel     (04/02/1994)
 *  - Loadable keymaps
 */

#include "kernel.h"
#include <termios.h>
#include <signal.h>
#include <unistd.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/keymap.h>
#include "tty.h"
#include "keymaps/us-std.src"

/* Standard and AT keyboard.  (PS/2 MCA implies AT throughout.) */
#define KEYBD           0x60    /* I/O port for keyboard data */

/* AT keyboard. */
#define KB_COMMAND      0x64    /* I/O port for commands on AT */
#define KB_GATE_A20     0x02    /* bit in output port to enable A20 line */
#define KB_PULSE_OUTPUT 0xF0    /* base for commands to pulse output port */
#define KB_RESET        0x01    /* bit in output port to reset CPU */
#define KB_STATUS       0x64    /* I/O port for status on AT */
#define KB_ACK          0xFA    /* keyboard ack response */
#define KB_BUSY         0x02    /* status bit set when KEYBD port ready */
#define LED_CODE        0xED    /* command to keyboard to set LEDs */
#define MAX_KB_ACK_RETRIES 0x1000       /* max #times to wait for kb ack */
#define MAX_KB_BUSY_RETRIES 0x1000      /* max #times to loop while kb busy */
#define KBIT            0x80    /* bit used to ack characters to keyboard */

/* Miscellaneous. */
#define ESC_SCAN           1    /* Reboot key when panicking */
#define SLASH_SCAN        53    /* to recognize numeric slash */
#define HOME_SCAN         71    /* first key on the numeric keypad */
#define DEL_SCAN          83    /* DEL for use in CTRL-ALT-DEL reboot */
#define CONSOLE            0    /* line number for console */
#define MEMCHECK_ADR   0x472    /* address to stop memory check after reboot */
#define MEMCHECK_MAG  0x1234    /* magic number to stop memory check */

#define kb_addr()       (&kb_lines[0])  /* there is only one keyboard */
#define KB_IN_BYTES       32    /* size of keyboard input buffer */

PRIVATE int alt1;               /* left alt key state */
PRIVATE int alt2;               /* right alt key state */
PRIVATE int capslock;           /* caps lock key state */
PRIVATE int esc;                /* escape scan code detected? */
PRIVATE int control;            /* control key state */
PRIVATE int caps_off;           /* 1 = normal position, 0 = depressed */
```

```c
PRIVATE int numlock;            /* number lock key state */
PRIVATE int num_off;            /* 1 = normal position, 0 = depressed */
PRIVATE int slock;              /* scroll lock key state */
PRIVATE int slock_off;          /* 1 = normal position, 0 = depressed */
PRIVATE int shift;              /* shift key state */

PRIVATE char numpad_map[] =
                {'H', 'Y', 'A', 'B', 'D', 'C', 'V', 'U', 'G', 'S', 'T', '@'};

/* Keyboard structure, 1 per console. */
struct kb_s {
  char *ihead;                  /* next free spot in input buffer */
  char *itail;                  /* scan code to return to TTY */
  int icount;                   /* # codes in buffer */
  char ibuf[KB_IN_BYTES];       /* input buffer */
};

PRIVATE struct kb_s kb_lines[NR_CONS];

FORWARD _PROTOTYPE( int kb_ack, (void) );
FORWARD _PROTOTYPE( int kb_wait, (void) );
FORWARD _PROTOTYPE( int func_key, (int scode) );
FORWARD _PROTOTYPE( int scan_keyboard, (void) );
FORWARD _PROTOTYPE( unsigned make_break, (int scode) );
FORWARD _PROTOTYPE( void set_leds, (void) );
FORWARD _PROTOTYPE( int kbd_hw_int, (int irq) );
FORWARD _PROTOTYPE( void kb_read, (struct tty *tp) );
FORWARD _PROTOTYPE( unsigned map_key, (int scode) );


/*===========================================================================*
 *                              map_key0                                     *
 *===========================================================================*/
/* Map a scan code to an ASCII code ignoring modifiers. */
#define map_key0(scode)  \
        ((unsigned) keymap[(scode) * MAP_COLS])


/*===========================================================================*
 *                              map_key                                      *
 *===========================================================================*/
PRIVATE unsigned map_key(scode)
int scode;
{
/* Map a scan code to an ASCII code. */

  int caps, column;
  u16_t *keyrow;

  if (scode == SLASH_SCAN && esc) return '/';   /* don't map numeric slash */

  keyrow = &keymap[scode * MAP_COLS];

  caps = shift;
  if (numlock && HOME_SCAN <= scode && scode <= DEL_SCAN) caps = !caps;
  if (capslock && (keyrow[0] & HASCAPS)) caps = !caps;

  if (alt1 || alt2) {
        column = 2;
        if (control || alt2) column = 3;        /* Ctrl + Alt1 == Alt2 */
```

```
                if (caps) column = 4;
        } else {
                column = 0;
                if (caps) column = 1;
                if (control) column = 5;
        }
        return keyrow[column] & ~HASCAPS;
}


/*===========================================================================*
 *                              kbd_hw_int                                   *
 *===========================================================================*/
PRIVATE int kbd_hw_int(irq)
int irq;
{
/* A keyboard interrupt has occurred.  Process it. */

  int code;
  unsigned km;
  register struct kb_s *kb;

  /* Fetch the character from the keyboard hardware and acknowledge it. */
  code = scan_keyboard();

  /* The IBM keyboard interrupts twice per key, once when depressed, once when
   * released.  Filter out the latter, ignoring all but the shift-type keys.
   * The shift-type keys 29, 42, 54, 56, 58, and 69 must be processed normally.
   */

  if (code & 0200) {
        /* A key has been released (high bit is set). */
        km = map_key0(code & 0177);
        if (km != CTRL && km != SHIFT && km != ALT && km != CALOCK
                        && km != NLOCK && km != SLOCK && km != EXTKEY)
                return 1;
  }

  /* Store the character in memory so the task can get at it later. */
  kb = kb_addr();
  if (kb->icount < KB_IN_BYTES) {
        *kb->ihead++ = code;
        if (kb->ihead == kb->ibuf + KB_IN_BYTES) kb->ihead = kb->ibuf;
        kb->icount++;
        tty_table[current].tty_events = 1;
        force_timeout();
  }
  /* Else it doesn't fit - discard it. */
  return 1;     /* Reenable keyboard interrupt */
}


/*==========================================================================*
 *                              kb_read                                     *
 *==========================================================================*/
PRIVATE void kb_read(tp)
tty_t *tp;
{
/* Process characters from the circular keyboard buffer. */

```

```
  struct kb_s *kb;
  char buf[3];
  int scode;
  unsigned ch;

  kb = kb_addr();
  tp = &tty_table[current];             /* always use the current console */

  while (kb->icount > 0) {
        scode = *kb->itail++;                   /* take one key scan code */
        if (kb->itail == kb->ibuf + KB_IN_BYTES) kb->itail = kb->ibuf;
        lock();
        kb->icount--;
        unlock();

        /* Function keys are being used for debug dumps. */
        if (func_key(scode)) continue;

        /* Perform make/break processing. */
        ch = make_break(scode);

        if (ch <= 0xFF) {
                /* A normal character. */
                buf[0] = ch;
                (void) in_process(tp, buf, 1);
        } else
        if (HOME <= ch && ch <= INSRT) {
                /* An ASCII escape sequence generated by the numeric pad. */
                buf[0] = ESC;
                buf[1] = '[';
                buf[2] = numpad_map[ch - HOME];
                (void) in_process(tp, buf, 3);
        } else
        if (ch == ALEFT) {
                /* Choose lower numbered console as current console. */
                select_console(current - 1);
        } else
        if (ch == ARIGHT) {
                /* Choose higher numbered console as current console. */
                select_console(current + 1);
        } else
        if (AF1 <= ch && ch <= AF12) {
                /* Alt-F1 is console, Alt-F2 is ttyc1, etc. */
                select_console(ch - AF1);
        }
  }
}


/*===========================================================================*
 *                              make_break                                   *
 *===========================================================================*/
PRIVATE unsigned make_break(scode)
int scode;                      /* scan code of key just struck or released */
{
/* This routine can handle keyboards that interrupt only on key depression,
 * as well as keyboards that interrupt on key depression and key release.
 * For efficiency, the interrupt routine filters out most key releases.
 */
  int ch, make;
```

```
  static int CAD_count = 0;

  /* Check for CTRL-ALT-DEL, and if found, halt the computer. This would
   * be better done in keyboard() in case TTY is hung, except control and
   * alt are set in the high level code.
   */
  if (control && (alt1 || alt2) && scode == DEL_SCAN)
  {
        if (++CAD_count == 3) wreboot(RBT_HALT);
        cause_sig(INIT_PROC_NR, SIGABRT);
        return -1;
  }

  /* High-order bit set on key release. */
  make = (scode & 0200 ? 0 : 1);        /* 0 = release, 1 = press */

  ch = map_key(scode & 0177);           /* map to ASCII */

  switch (ch) {
        case CTRL:
                control = make;
                ch = -1;
                break;
        case SHIFT:
                shift = make;
                ch = -1;
                break;
        case ALT:
                if (make) {
                        if (esc) alt2 = 1; else alt1 = 1;
                } else {
                        alt1 = alt2 = 0;
                }
                ch = -1;
                break;
        case CALOCK:
                if (make && caps_off) {
                        capslock = 1 - capslock;
                        set_leds();
                }
                caps_off = 1 - make;
                ch = -1;
                break;
        case NLOCK:
                if (make && num_off) {
                        numlock = 1 - numlock;
                        set_leds();
                }
                num_off = 1 - make;
                ch = -1;
                break;
        case SLOCK:
                if (make & slock_off) {
                        slock = 1 - slock;
                        set_leds();
                }
                slock_off = 1 - make;
                ch = -1;
                break;
        case EXTKEY:
```

```
                esc = 1;
                return(-1);
        default:
                if (!make) ch = -1;
  }
  esc = 0;
  return(ch);
}


/*===========================================================================*
 *                              set_leds                                     *
 *===========================================================================*/
PRIVATE void set_leds()
{
/* Set the LEDs on the caps lock and num lock keys */

  unsigned leds;

  if (!pc_at) return;   /* PC/XT doesn't have LEDs */

  /* encode LED bits */
  leds = (slock << 0) | (numlock << 1) | (capslock << 2);

  kb_wait();                    /* wait for buffer empty  */
  out_byte(KEYBD, LED_CODE);    /* prepare keyboard to accept LED values */
  kb_ack();                     /* wait for ack response  */

  kb_wait();                    /* wait for buffer empty  */
  out_byte(KEYBD, leds);        /* give keyboard LED values */
  kb_ack();                     /* wait for ack response  */
}


/*==========================================================================*
 *                              kb_wait                                     *
 *==========================================================================*/
PRIVATE int kb_wait()
{
/* Wait until the controller is ready; return zero if this times out. */

  int retries;

  retries = MAX_KB_BUSY_RETRIES + 1;
  while (--retries != 0 && in_byte(KB_STATUS) & KB_BUSY)
        ;                       /* wait until not busy */
  return(retries);              /* nonzero if ready */
}


/*==========================================================================*
 *                              kb_ack                                      *
 *==========================================================================*/
PRIVATE int kb_ack()
{
/* Wait until kbd acknowledges last command; return zero if this times out. */

  int retries;

  retries = MAX_KB_ACK_RETRIES + 1;
```

```
  while (--retries != 0 && in_byte(KEYBD) != KB_ACK)
        ;                       /* wait for ack */
  return(retries);              /* nonzero if ack received */
}


/*===========================================================================*
 *                              kb_init                                      *
 *===========================================================================*/
PUBLIC void kb_init(tp)
tty_t *tp;
{
/* Initialize the keyboard driver. */

  register struct kb_s *kb;

  /* Input function. */
  tp->tty_devread = kb_read;

  kb = kb_addr();

  /* Set up input queue. */
  kb->ihead = kb->itail = kb->ibuf;

  /* Set initial values. */
  caps_off = 1;
  num_off = 1;
  slock_off = 1;
  esc = 0;

  set_leds();                   /* turn off numlock led */

  scan_keyboard();              /* stop lockup from leftover keystroke */

  put_irq_handler(KEYBOARD_IRQ, kbd_hw_int);    /* set the interrupt handler */
  enable_irq(KEYBOARD_IRQ);     /* safe now everything initialised! */
}


/*===========================================================================*
 *                              kbd_loadmap                                  *
 *===========================================================================*/
PUBLIC int kbd_loadmap(user_phys)
phys_bytes user_phys;
{
/* Load a new keymap. */

  phys_copy(user_phys, vir2phys(keymap), (phys_bytes) sizeof(keymap));
  return(OK);
}


/*===========================================================================*
 *                              func_key                                     *
 *===========================================================================*/
PRIVATE int func_key(scode)
int scode;                      /* scan code for a function key */
{
/* This procedure traps function keys for debugging and control purposes. */

```

```
  unsigned code;

  code = map_key0(scode);                       /* first ignore modifiers */
  if (code < F1 || code > F12) return(FALSE);   /* not our job */

  switch (map_key(scode)) {                     /* include modifiers */

  case F1:      p_dmp(); break;         /* print process table */
  case F2:      map_dmp(); break;       /* print memory map */
  case F3:      toggle_scroll(); break; /* hardware vs. software scrolling */
  case CF7:     sigchar(&tty_table[CONSOLE], SIGQUIT); break;
  case CF8:     sigchar(&tty_table[CONSOLE], SIGINT); break;
  case CF9:     sigchar(&tty_table[CONSOLE], SIGKILL); break;
  default:      return(FALSE);
  }
  return(TRUE);
}


/*===========================================================================*
 *                              scan_keyboard                                *
 *===========================================================================*/
PRIVATE int scan_keyboard()
{
/* Fetch the character from the keyboard hardware and acknowledge it. */

  int code;
  int val;

  code = in_byte(KEYBD);        /* get the scan code for the key struck */
  val = in_byte(PORT_B);        /* strobe the keyboard to ack the char */
  out_byte(PORT_B, val | KBIT); /* strobe the bit high */
  out_byte(PORT_B, val);        /* now strobe it low */
  return code;
}


/*===========================================================================*
 *                              wreboot                                      *
 *===========================================================================*/
PUBLIC void wreboot(how)
int how;                /* 0 = halt, 1 = reboot, 2 = panic!, ... */
{
/* Wait for keystrokes for printing debugging info and reboot. */

  int quiet, code;
  static u16_t magic = MEMCHECK_MAG;
  struct tasktab *ttp;

  /* Mask all interrupts. */
  out_byte(INT_CTLMASK, ~0);

  /* Tell several tasks to stop. */
  cons_stop();
  floppy_stop();
  clock_stop();

  if (how == RBT_HALT) {
        printf("System Halted\n");
        if (!mon_return) how = RBT_PANIC;
```

```
  }

  if (how == RBT_PANIC) {
        /* A panic! */
        printf("Hit ESC to reboot, F-keys for debug dumps\n");

        (void) scan_keyboard(); /* ack any old input */
        quiet = scan_keyboard();/* quiescent value (0 on PC, last code on AT)*/
        for (;;) {
                milli_delay(100);       /* pause for a decisecond */
                code = scan_keyboard();
                if (code != quiet) {
                        /* A key has been pressed. */
                        if (code == ESC_SCAN) break; /* reboot if ESC typed */
                        (void) func_key(code);       /* process function key */
                        quiet = scan_keyboard();
                }
        }
        how = RBT_REBOOT;
  }

  if (how == RBT_REBOOT) printf("Rebooting\n");

  if (mon_return && how != RBT_RESET) {
        /* Reinitialize the interrupt controllers to the BIOS defaults. */
        intr_init(0);
        out_byte(INT_CTLMASK, 0);
        out_byte(INT2_CTLMASK, 0);

        /* Return to the boot monitor. */
        if (how == RBT_HALT) {
                reboot_code = vir2phys("");
        } else
        if (how == RBT_REBOOT) {
                reboot_code = vir2phys("delay;boot");
        }
        level0(monitor);
  }

  /* Stop BIOS memory test. */
  phys_copy(vir2phys(&magic), (phys_bytes) MEMCHECK_ADR,
                                                (phys_bytes) sizeof(magic));

  if (protected_mode) {
        /* Use the AT keyboard controller to reset the processor.
         * The A20 line is kept enabled in case this code is ever
         * run from extended memory, and because some machines
         * appear to drive the fake A20 high instead of low just
         * after reset, leading to an illegal opode trap.  This bug
         * is more of a problem if the fake A20 is in use, as it
         * would be if the keyboard reset were used for real mode.
         */
        kb_wait();
        out_byte(KB_COMMAND,
                 KB_PULSE_OUTPUT | (0x0F & ~(KB_GATE_A20 | KB_RESET)));
        milli_delay(10);

        /* If the nice method fails then do a reset.  In protected
         * mode this means a processor shutdown.
         */
```

```
                printf("Hard reset...\n");
                milli_delay(250);
        }
        /* In real mode, jumping to the reset address is good enough. */
        level0(reset);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/console.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Code and data for the IBM console driver.
 *
 * The 6845 video controller used by the IBM PC shares its video memory with
 * the CPU somewhere in the 0xB0000 memory bank.  To the 6845 this memory
 * consists of 16-bit words.  Each word has a character code in the low byte
 * and a so-called attribute byte in the high byte.  The CPU directly modifies
 * video memory to display characters, and sets two registers on the 6845 that
 * specify the video origin and the cursor position.  The video origin is the
 * place in video memory where the first character (upper left corner) can
 * be found.  Moving the origin is a fast way to scroll the screen.  Some
 * video adapters wrap around the top of video memory, so the origin can
 * move without bounds.  For other adapters screen memory must sometimes be
 * moved to reset the origin.  All computations on video memory use character
 * (word) addresses for simplicity and assume there is no wrapping.  The
 * assembly support functions translate the word addresses to byte addresses
 * and the scrolling function worries about wrapping.
 */

#include "kernel.h"
#include <termios.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "protect.h"
#include "tty.h"
#include "proc.h"

/* Definitions used by the console driver. */
#define MONO_BASE    0xB0000L   /* base of mono video memory */
#define COLOR_BASE   0xB8000L   /* base of color video memory */
#define MONO_SIZE     0x1000    /* 4K mono video memory */
#define COLOR_SIZE    0x4000    /* 16K color video memory */
#define EGA_SIZE      0x8000    /* EGA & VGA have at least 32K */
#define BLANK_COLOR   0x0700    /* determines cursor color on blank screen */
#define SCROLL_UP          0    /* scroll forward */
#define SCROLL_DOWN        1    /* scroll backward */
#define BLANK_MEM ((u16_t *) 0) /* tells mem_vid_copy() to blank the screen */
#define CONS_RAM_WORDS    80    /* video ram buffer size */
#define MAX_ESC_PARMS      2    /* number of escape sequence params allowed */

/* Constants relating to the controller chips. */
#define M_6845         0x3B4    /* port for 6845 mono */
#define C_6845         0x3D4    /* port for 6845 color */
#define EGA            0x3C4    /* port for EGA or VGA card */
#define INDEX              0    /* 6845's index register */
#define DATA               1    /* 6845's data register */
```

```
#define VID_ORG         12    /* 6845's origin register */
#define CURSOR          14    /* 6845's cursor register */

/* Beeper. */
#define BEEP_FREQ     0x0533    /* value to put into timer to set beep freq */
#define B_TIME             3    /* length of CTRL-G beep is ticks */

/* definitions used for font management */
#define GA_SEQUENCER_INDEX      0x3C4
#define GA_SEQUENCER_DATA       0x3C5
#define GA_GRAPHICS_INDEX       0x3CE
#define GA_GRAPHICS_DATA        0x3CF
#define GA_VIDEO_ADDRESS        0xA0000L
#define GA_FONT_SIZE            8192

/* Global variables used by the console driver. */
PUBLIC unsigned vid_seg;        /* video ram selector (0xB0000 or 0xB8000) */
PUBLIC unsigned vid_size;       /* 0x2000 for color or 0x0800 for mono */
PUBLIC unsigned vid_mask;       /* 0x1FFF for color or 0x07FF for mono */
PUBLIC unsigned blank_color = BLANK_COLOR; /* display code for blank */

/* Private variables used by the console driver. */
PRIVATE int vid_port;           /* I/O port for accessing 6845 */
PRIVATE int wrap;               /* hardware can wrap? */
PRIVATE int softscroll;         /* 1 = software scrolling, 0 = hardware */
PRIVATE unsigned vid_base;      /* base of video ram (0xB000 or 0xB800) */
PRIVATE int beeping;            /* speaker is beeping? */
#define scr_width       80      /* # characters on a line */
#define scr_lines       25      /* # lines on the screen */
#define scr_size        (80*25) /* # characters on the screen */

/* Per console data. */
typedef struct console {
  tty_t *c_tty;                 /* associated TTY struct */
  int c_column;                 /* current column number (0-origin) */
  int c_row;                    /* current row (0 at top of screen) */
  int c_rwords;                 /* number of WORDS (not bytes) in outqueue */
  unsigned c_start;             /* start of video memory of this console */
  unsigned c_limit;             /* limit of this console's video memory */
  unsigned c_org;               /* location in RAM where 6845 base points */
  unsigned c_cur;               /* current position of cursor in video RAM */
  unsigned c_attr;              /* character attribute */
  unsigned c_blank;             /* blank attribute */
  char c_esc_state;             /* 0=normal, 1=ESC, 2=ESC[ */
  char c_esc_intro;             /* Distinguishing character following ESC */
  int *c_esc_parmp;             /* pointer to current escape parameter */
  int c_esc_parmv[MAX_ESC_PARMS];       /* list of escape parameters */
  u16_t c_ramqueue[CONS_RAM_WORDS];     /* buffer for video RAM */
} console_t;

PRIVATE int nr_cons= 1;         /* actual number of consoles */
PRIVATE console_t cons_table[NR_CONS];
PRIVATE console_t *curcons;     /* currently visible */

/* Color if using a color controller. */
#define color   (vid_port == C_6845)

/* Map from ANSI colors to the attributes used by the PC */
PRIVATE int ansi_colors[8] = {0, 4, 2, 6, 1, 5, 3, 7};

```

```
/* Structure used for font management */
struct sequence {
        unsigned short index;
        unsigned char port;
        unsigned char value;
};

FORWARD _PROTOTYPE( void cons_write, (struct tty *tp)                   );
FORWARD _PROTOTYPE( void cons_echo, (tty_t *tp, int c)                  );
FORWARD _PROTOTYPE( void out_char, (console_t *cons, int c)             );
FORWARD _PROTOTYPE( void beep, (void)                                   );
FORWARD _PROTOTYPE( void do_escape, (console_t *cons, int c)            );
FORWARD _PROTOTYPE( void flush, (console_t *cons)                       );
FORWARD _PROTOTYPE( void parse_escape, (console_t *cons, int c)         );
FORWARD _PROTOTYPE( void scroll_screen, (console_t *cons, int dir)      );
FORWARD _PROTOTYPE( void set_6845, (int reg, unsigned val)              );
FORWARD _PROTOTYPE( void stop_beep, (void)                              );
FORWARD _PROTOTYPE( void cons_org0, (void)                              );
FORWARD _PROTOTYPE( void ga_program, (struct sequence *seq) );


/*===========================================================================*
 *                              cons_write                                   *
 *===========================================================================*/
PRIVATE void cons_write(tp)
register struct tty *tp;        /* tells which terminal is to be used */
{
/* Copy as much data as possible to the output queue, then start I/O.  On
 * memory-mapped terminals, such as the IBM console, the I/O will also be
 * finished, and the counts updated.  Keep repeating until all I/O done.
 */

  int count;
  register char *tbuf;
  char buf[64];
  phys_bytes user_phys;
  console_t *cons = tp->tty_priv;

  /* Check quickly for nothing to do, so this can be called often without
   * unmodular tests elsewhere.
   */
  if ((count = tp->tty_outleft) == 0 || tp->tty_inhibited) return;

  /* Copy the user bytes to buf[] for decent addressing. Loop over the
   * copies, since the user buffer may be much larger than buf[].
   */
  do {
        if (count > sizeof(buf)) count = sizeof(buf);
        user_phys = proc_vir2phys(proc_addr(tp->tty_outproc), tp->tty_out_vir);
        phys_copy(user_phys, vir2phys(buf), (phys_bytes) count);
        tbuf = buf;

        /* Update terminal data structure. */
        tp->tty_out_vir += count;
        tp->tty_outcum += count;
        tp->tty_outleft -= count;

        /* Output each byte of the copy to the screen.  Avoid calling
         * out_char() for the "easy" characters, put them into the buffer
         * directly.
```

```
	 */
	do {
		if ((unsigned) *tbuf < ' ' || cons->c_esc_state > 0
			|| cons->c_column >= scr_width
			|| cons->c_rwords >= buflen(cons->c_ramqueue))
		{
			out_char(cons, *tbuf++);
		} else {
			cons->c_ramqueue[cons->c_rwords++] =
					cons->c_attr | (*tbuf++ & BYTE);
			cons->c_column++;
		}
	} while (--count != 0);
  } while ((count = tp->tty_outleft) != 0 && !tp->tty_inhibited);

  flush(cons);			/* transfer anything buffered to the screen */

  /* Reply to the writer if all output is finished. */
  if (tp->tty_outleft == 0) {
	tty_reply(tp->tty_outrepcode, tp->tty_outcaller, tp->tty_outproc,
							tp->tty_outcum);
	tp->tty_outcum = 0;
  }
}


/*===========================================================================*
 *				cons_echo				     *
 *===========================================================================*/
PRIVATE void cons_echo(tp, c)
register tty_t *tp;		/* pointer to tty struct */
int c;				/* character to be echoed */
{
/* Echo keyboard input (print & flush). */
  console_t *cons = tp->tty_priv;

  out_char(cons, c);
  flush(cons);
}


/*===========================================================================*
 *				out_char				     *
 *===========================================================================*/
PRIVATE void out_char(cons, c)
register console_t *cons;	/* pointer to console struct */
int c;				/* character to be output */
{
/* Output a character on the console.  Check for escape sequences first. */
  if (cons->c_esc_state > 0) {
	parse_escape(cons, c);
	return;
  }

  switch(c) {
	case 000:		/* null is typically used for padding */
		return;		/* better not do anything */

	case 007:		/* ring the bell */
		flush(cons);	/* print any chars queued for output */
```

```
                beep();
                return;

        case '\b':              /* backspace */
                if (--cons->c_column < 0) {
                        if (--cons->c_row >= 0) cons->c_column += scr_width;
                }
                flush(cons);
                return;

        case '\n':              /* line feed */
                if ((cons->c_tty->tty_termios.c_oflag & (OPOST|ONLCR))
                                                == (OPOST|ONLCR)) {
                        cons->c_column = 0;
                }
                /*FALL THROUGH*/
        case 013:               /* CTRL-K */
        case 014:               /* CTRL-L */
                if (cons->c_row == scr_lines-1) {
                        scroll_screen(cons, SCROLL_UP);
                } else {
                        cons->c_row++;
                }
                flush(cons);
                return;

        case '\r':              /* carriage return */
                cons->c_column = 0;
                flush(cons);
                return;

        case '\t':              /* tab */
                cons->c_column = (cons->c_column + TAB_SIZE) & ~TAB_MASK;
                if (cons->c_column > scr_width) {
                        cons->c_column -= scr_width;
                        if (cons->c_row == scr_lines-1) {
                                scroll_screen(cons, SCROLL_UP);
                        } else {
                                cons->c_row++;
                        }
                }
                flush(cons);
                return;

        case 033:               /* ESC - start of an escape sequence */
                flush(cons);    /* print any chars queued for output */
                cons->c_esc_state = 1;  /* mark ESC as seen */
                return;

        default:                /* printable chars are stored in ramqueue */
                if (cons->c_column >= scr_width) {
                        if (!LINEWRAP) return;
                        if (cons->c_row == scr_lines-1) {
                                scroll_screen(cons, SCROLL_UP);
                        } else {
                                cons->c_row++;
                        }
                        cons->c_column = 0;
                        flush(cons);
                }
```

```
                if (cons->c_rwords == buflen(cons->c_ramqueue)) flush(cons);
                cons->c_ramqueue[cons->c_rwords++] = cons->c_attr | (c & BYTE);
                cons->c_column++;                       /* next column */
                return;
  }
}


/*===========================================================================*
 *                              scroll_screen                                *
 *===========================================================================*/
PRIVATE void scroll_screen(cons, dir)
register console_t *cons;       /* pointer to console struct */
int dir;                        /* SCROLL_UP or SCROLL_DOWN */
{
  unsigned new_line, new_org, chars;

  flush(cons);
  chars = scr_size - scr_width;         /* one screen minus one line */

  /* Scrolling the screen is a real nuisance due to the various incompatible
   * video cards.  This driver supports software scrolling (Hercules?),
   * hardware scrolling (mono and CGA cards) and hardware scrolling without
   * wrapping (EGA and VGA cards).  In the latter case we must make sure that
   *            c_start <= c_org && c_org + scr_size <= c_limit
   * holds, because EGA and VGA don't wrap around the end of video memory.
   */
  if (dir == SCROLL_UP) {
        /* Scroll one line up in 3 ways: soft, avoid wrap, use origin. */
        if (softscroll) {
                vid_vid_copy(cons->c_start + scr_width, cons->c_start, chars);
        } else
        if (!wrap && cons->c_org + scr_size + scr_width >= cons->c_limit) {
                vid_vid_copy(cons->c_org + scr_width, cons->c_start, chars);
                cons->c_org = cons->c_start;
        } else {
                cons->c_org = (cons->c_org + scr_width) & vid_mask;
        }
        new_line = (cons->c_org + chars) & vid_mask;
  } else {
        /* Scroll one line down in 3 ways: soft, avoid wrap, use origin. */
        if (softscroll) {
                vid_vid_copy(cons->c_start, cons->c_start + scr_width, chars);
        } else
        if (!wrap && cons->c_org < cons->c_start + scr_width) {
                new_org = cons->c_limit - scr_size;
                vid_vid_copy(cons->c_org, new_org + scr_width, chars);
                cons->c_org = new_org;
        } else {
                cons->c_org = (cons->c_org - scr_width) & vid_mask;
        }
        new_line = cons->c_org;
  }
  /* Blank the new line at top or bottom. */
  blank_color = cons->c_blank;
  mem_vid_copy(BLANK_MEM, new_line, scr_width);

  /* Set the new video origin. */
  if (cons == curcons) set_6845(VID_ORG, cons->c_org);
  flush(cons);
```

```
}


/*===========================================================================*
 *                              flush                                        *
 *===========================================================================*/
PRIVATE void flush(cons)
register console_t *cons;       /* pointer to console struct */
{
/* Send characters buffered in 'ramqueue' to screen memory, check the new
 * cursor position, compute the new hardware cursor position and set it.
 */
  unsigned cur;
  tty_t *tp = cons->c_tty;

  /* Have the characters in 'ramqueue' transferred to the screen. */
  if (cons->c_rwords > 0) {
        mem_vid_copy(cons->c_ramqueue, cons->c_cur, cons->c_rwords);
        cons->c_rwords = 0;

        /* TTY likes to know the current column and if echoing messed up. */
        tp->tty_position = cons->c_column;
        tp->tty_reprint = TRUE;
  }

  /* Check and update the cursor position. */
  if (cons->c_column < 0) cons->c_column = 0;
  if (cons->c_column > scr_width) cons->c_column = scr_width;
  if (cons->c_row < 0) cons->c_row = 0;
  if (cons->c_row >= scr_lines) cons->c_row = scr_lines - 1;
  cur = cons->c_org + cons->c_row * scr_width + cons->c_column;
  if (cur != cons->c_cur) {
        if (cons == curcons) set_6845(CURSOR, cur);
        cons->c_cur = cur;
  }
}


/*===========================================================================*
 *                              parse_escape                                 *
 *===========================================================================*/
PRIVATE void parse_escape(cons, c)
register console_t *cons;       /* pointer to console struct */
char c;                         /* next character in escape sequence */
{
/* The following ANSI escape sequences are currently supported.
 * If n and/or m are omitted, they default to 1. Omitted s defaults to 0.
 *   ESC [nA moves up n lines
 *   ESC [nB moves down n lines
 *   ESC [nC moves right n spaces
 *   ESC [nD moves left n spaces
 *   ESC [m;nH moves cursor to (m,n)
 *   ESC [sJ clears screen relative to cursor (0 to end, 1 from start, 2 all)
 *   ESC [sK clears line relative to cursor (0 to end, 1 from start, 2 all)
 *   ESC [nL inserts n lines at cursor
 *   ESC [nM deletes n lines at cursor
 *   ESC [nP deletes n chars at cursor
 *   ESC [n@ inserts n chars at cursor
 *   ESC [nm enables rendition n (0= normal, 1=bold, 4=underline, 5=blinking,
 *       7=reverse, 30..37 set foreground color, 40..47 set background color)
```

```
 *   ESC M scrolls the screen backwards if the cursor is on the top line
 */

  switch (cons->c_esc_state) {
    case 1:                     /* ESC seen */
        cons->c_esc_intro = '\0';
        cons->c_esc_parmp = cons->c_esc_parmv;
        cons->c_esc_parmv[0] = cons->c_esc_parmv[1] = 0;
        switch (c) {
            case '[':   /* Control Sequence Introducer */
                cons->c_esc_intro = c;
                cons->c_esc_state = 2;
                break;
            case 'M':   /* Reverse Index */
                do_escape(cons, c);
                break;
            default:
                cons->c_esc_state = 0;
        }
        break;

    case 2:                     /* ESC [ seen */
        if (c >= '0' && c <= '9') {
                if (cons->c_esc_parmp < bufend(cons->c_esc_parmv))
                        *cons->c_esc_parmp = *cons->c_esc_parmp * 10 + (c-'0');
        } else
        if (c == ';') {
                if (++cons->c_esc_parmp < bufend(cons->c_esc_parmv))
                        *cons->c_esc_parmp = 0;
        } else {
                do_escape(cons, c);
        }
        break;
  }
}


/*===========================================================================*
 *                              do_escape                                    *
 *===========================================================================*/
PRIVATE void do_escape(cons, c)
register console_t *cons;       /* pointer to console struct */
char c;                         /* next character in escape sequence */
{
  int value, n;
  unsigned src, dst, count;

  /* Some of these things hack on screen RAM, so it had better be up to date */
  flush(cons);

  if (cons->c_esc_intro == '\0') {
        /* Handle a sequence beginning with just ESC */
        switch (c) {
            case 'M':           /* Reverse Index */
                if (cons->c_row == 0) {
                        scroll_screen(cons, SCROLL_DOWN);
                } else {
                        cons->c_row--;
                }
                flush(cons);
```

```
                  break;

              default: break;
          }
    } else
    if (cons->c_esc_intro == ’[’) {
          /* Handle a sequence beginning with ESC [ and parameters */
          value = cons->c_esc_parmv[0];
          switch (c) {
              case ’A’:           /* ESC [nA moves up n lines */
                  n = (value == 0 ? 1 : value);
                  cons->c_row -= n;
                  flush(cons);
                  break;

              case ’B’:           /* ESC [nB moves down n lines */
                  n = (value == 0 ? 1 : value);
                  cons->c_row += n;
                  flush(cons);
                  break;

              case ’C’:           /* ESC [nC moves right n spaces */
                  n = (value == 0 ? 1 : value);
                  cons->c_column += n;
                  flush(cons);
                  break;

              case ’D’:           /* ESC [nD moves left n spaces */
                  n = (value == 0 ? 1 : value);
                  cons->c_column -= n;
                  flush(cons);
                  break;

              case ’H’:           /* ESC [m;nH" moves cursor to (m,n) */
                  cons->c_row = cons->c_esc_parmv[0] - 1;
                  cons->c_column = cons->c_esc_parmv[1] - 1;
                  flush(cons);
                  break;

              case ’J’:           /* ESC [sJ clears in display */
                  switch (value) {
                      case 0:     /* Clear from cursor to end of screen */
                          count = scr_size - (cons->c_cur - cons->c_org);
                          dst = cons->c_cur;
                          break;
                      case 1:     /* Clear from start of screen to cursor */
                          count = cons->c_cur - cons->c_org;
                          dst = cons->c_org;
                          break;
                      case 2:     /* Clear entire screen */
                          count = scr_size;
                          dst = cons->c_org;
                          break;
                      default:    /* Do nothing */
                          count = 0;
                          dst = cons->c_org;
                  }
                  blank_color = cons->c_blank;
                  mem_vid_copy(BLANK_MEM, dst, count);
                  break;
```

```
	    case 'K':		/* ESC [sK clears line from cursor */
		switch (value) {
		    case 0:	/* Clear from cursor to end of line */
			count = scr_width - cons->c_column;
			dst = cons->c_cur;
			break;
		    case 1:	/* Clear from beginning of line to cursor */
			count = cons->c_column;
			dst = cons->c_cur - cons->c_column;
			break;
		    case 2:	/* Clear entire line */
			count = scr_width;
			dst = cons->c_cur - cons->c_column;
			break;
		    default:	/* Do nothing */
			count = 0;
			dst = cons->c_cur;
		}
		blank_color = cons->c_blank;
		mem_vid_copy(BLANK_MEM, dst, count);
		break;

	    case 'L':		/* ESC [nL inserts n lines at cursor */
		n = value;
		if (n < 1) n = 1;
		if (n > (scr_lines - cons->c_row))
			n = scr_lines - cons->c_row;

		src = cons->c_org + cons->c_row * scr_width;
		dst = src + n * scr_width;
		count = (scr_lines - cons->c_row - n) * scr_width;
		vid_vid_copy(src, dst, count);
		blank_color = cons->c_blank;
		mem_vid_copy(BLANK_MEM, src, n * scr_width);
		break;

	    case 'M':		/* ESC [nM deletes n lines at cursor */
		n = value;
		if (n < 1) n = 1;
		if (n > (scr_lines - cons->c_row))
			n = scr_lines - cons->c_row;

		dst = cons->c_org + cons->c_row * scr_width;
		src = dst + n * scr_width;
		count = (scr_lines - cons->c_row - n) * scr_width;
		vid_vid_copy(src, dst, count);
		blank_color = cons->c_blank;
		mem_vid_copy(BLANK_MEM, dst + count, n * scr_width);
		break;

	    case '@':		/* ESC [n@ inserts n chars at cursor */
		n = value;
		if (n < 1) n = 1;
		if (n > (scr_width - cons->c_column))
			n = scr_width - cons->c_column;

		src = cons->c_cur;
		dst = src + n;
		count = scr_width - cons->c_column - n;
```

```
            vid_vid_copy(src, dst, count);
            blank_color = cons->c_blank;
            mem_vid_copy(BLANK_MEM, src, n);
            break;

        case ’P’:           /* ESC [nP deletes n chars at cursor */
            n = value;
            if (n < 1) n = 1;
            if (n > (scr_width - cons->c_column))
                    n = scr_width - cons->c_column;

            dst = cons->c_cur;
            src = dst + n;
            count = scr_width - cons->c_column - n;
            vid_vid_copy(src, dst, count);
            blank_color = cons->c_blank;
            mem_vid_copy(BLANK_MEM, dst + count, n);
            break;

        case ’m’:           /* ESC [nm enables rendition n */
            switch (value) {
                case 1:     /* BOLD  */
                    if (color) {
                            /* Can’t do bold, so use yellow */
                            cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0E00;
                    } else {
                            /* Set intensity bit */
                            cons->c_attr |= 0x0800;
                    }
                    break;

                case 4:     /* UNDERLINE */
                    if (color) {
                            /* Use light green */
                            cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0A00;
                    } else {
                            cons->c_attr = (cons->c_attr & 0x8900);
                    }
                    break;

                case 5:     /* BLINKING */
                    if (color) {
                            /* Use magenta */
                            cons->c_attr = (cons->c_attr & 0xf0ff) | 0x0500;
                    } else {
                            /* Set the blink bit */
                            cons->c_attr |= 0x8000;
                    }
                    break;

                case 7:     /* REVERSE */
                    if (color) {
                            /* Swap fg and bg colors */
                            cons->c_attr =
                                    ((cons->c_attr & 0xf000) >> 4) |
                                    ((cons->c_attr & 0x0f00) << 4);
                    } else
                    if ((cons->c_attr & 0x7000) == 0) {
                            cons->c_attr = (cons->c_attr & 0x8800) | 0x7000;
                    } else {
```

```
                                cons->c_attr = (cons->c_attr & 0x8800) | 0x0700;
                        }
                        break;

                    default:    /* COLOR */
                        if (30 <= value && value <= 37) {
                                cons->c_attr =
                                        (cons->c_attr & 0xf0ff) |
                                        (ansi_colors[(value - 30)] << 8);
                                cons->c_blank =
                                        (cons->c_blank & 0xf0ff) |
                                        (ansi_colors[(value - 30)] << 8);
                        } else
                        if (40 <= value && value <= 47) {
                                cons->c_attr =
                                        (cons->c_attr & 0x0fff) |
                                        (ansi_colors[(value - 40)] << 12);
                                cons->c_blank =
                                        (cons->c_blank & 0x0fff) |
                                        (ansi_colors[(value - 40)] << 12);
                        } else {
                                cons->c_attr = cons->c_blank;
                        }
                        break;
                }
                break;
        }
  }
  cons->c_esc_state = 0;
}


/*===========================================================================*
 *                              set_6845                                     *
 *===========================================================================*/
PRIVATE void set_6845(reg, val)
int reg;                        /* which register pair to set */
unsigned val;                   /* 16-bit value to set it to */
{
/* Set a register pair inside the 6845.
 * Registers 12-13 tell the 6845 where in video ram to start
 * Registers 14-15 tell the 6845 where to put the cursor
 */
  lock();                       /* try to stop h/w loading in-between value */
  out_byte(vid_port + INDEX, reg);              /* set the index register */
  out_byte(vid_port + DATA, (val>>8) & BYTE);   /* output high byte */
  out_byte(vid_port + INDEX, reg + 1);          /* again */
  out_byte(vid_port + DATA, val&BYTE);          /* output low byte */
  unlock();
}


/*===========================================================================*
 *                              beep                                         *
 *===========================================================================*/
PRIVATE void beep()
{
/* Making a beeping sound on the speaker (output for CRTL-G).
 * This routine works by turning on the bits 0 and 1 in port B of the 8255
 * chip that drives the speaker.
```

```
 */

  message mess;

  if (beeping) return;
  out_byte(TIMER_MODE, 0xB6);   /* set up timer channel 2 (square wave) */
  out_byte(TIMER2, BEEP_FREQ & BYTE);   /* load low-order bits of frequency */
  out_byte(TIMER2, (BEEP_FREQ >> 8) & BYTE);    /* now high-order bits */
  lock();                       /* guard PORT_B from keyboard intr handler */
  out_byte(PORT_B, in_byte(PORT_B) | 3);        /* turn on beep bits */
  unlock();
  beeping = TRUE;

  mess.m_type = SET_ALARM;
  mess.CLOCK_PROC_NR = TTY;
  mess.DELTA_TICKS = B_TIME;
  mess.FUNC_TO_CALL = (sighandler_t) stop_beep;
  sendrec(CLOCK, &mess);
}


/*===========================================================================*
 *                              stop_beep                                    *
 *===========================================================================*/
PRIVATE void stop_beep()
{
/* Turn off the beeper by turning off bits 0 and 1 in PORT_B. */

  lock();                       /* guard PORT_B from keyboard intr handler */
  out_byte(PORT_B, in_byte(PORT_B) & ~3);
  beeping = FALSE;
  unlock();
}


/*===========================================================================*
 *                              scr_init                                     *
 *===========================================================================*/
PUBLIC void scr_init(tp)
tty_t *tp;
{
/* Initialize the screen driver. */
  console_t *cons;
  phys_bytes vid_base;
  u16_t bios_crtbase;
  int line;
  unsigned page_size;

  /* Associate console and TTY. */
  line = tp - &tty_table[0];
  if (line >= nr_cons) return;
  cons = &cons_table[line];
  cons->c_tty = tp;
  tp->tty_priv = cons;

  /* Initialize the keyboard driver. */
  kb_init(tp);

  /* Output functions. */
  tp->tty_devwrite = cons_write;
```

```
  tp->tty_echo = cons_echo;

  /* Get the BIOS parameters that tells the VDU I/O base register. */
  phys_copy(0x463L, vir2phys(&bios_crtbase), 2L);

  vid_port = bios_crtbase;

  if (color) {
        vid_base = COLOR_BASE;
        vid_size = COLOR_SIZE;
  } else {
        vid_base = MONO_BASE;
        vid_size = MONO_SIZE;
  }
  if (ega) vid_size = EGA_SIZE; /* for both EGA and VGA */
  wrap = !ega;

  vid_seg = protected_mode ? VIDEO_SELECTOR : physb_to_hclick(vid_base);
  init_dataseg(&gdt[VIDEO_INDEX], vid_base, (phys_bytes) vid_size,
                                                        TASK_PRIVILEGE);
  vid_size >>= 1;               /* word count */
  vid_mask = vid_size - 1;

  /* There can be as many consoles as video memory allows. */
  nr_cons = vid_size / scr_size;
  if (nr_cons > NR_CONS) nr_cons = NR_CONS;
  if (nr_cons > 1) wrap = 0;
  page_size = vid_size / nr_cons;
  cons->c_start = line * page_size;
  cons->c_limit = cons->c_start + page_size;
  cons->c_org = cons->c_start;
  cons->c_attr = cons->c_blank = BLANK_COLOR;

  /* Clear the screen. */
  blank_color = BLANK_COLOR;
  mem_vid_copy(BLANK_MEM, cons->c_start, scr_size);
  select_console(0);
}


/*===========================================================================*
 *                              putk                                         *
 *===========================================================================*/
PUBLIC void putk(c)
int c;                          /* character to print */
{
/* This procedure is used by the version of printf() that is linked with
 * the kernel itself.  The one in the library sends a message to FS, which is
 * not what is needed for printing within the kernel.  This version just queues
 * the character and starts the output.
 */

  if (c != 0) {
        if (c == '\n') putk('\r');
        out_char(&cons_table[0], (int) c);
  } else {
        flush(&cons_table[0]);
  }
}
```

```c
/*===========================================================================*
 *                              toggle_scroll                                *
 *===========================================================================*/
PUBLIC void toggle_scroll()
{
/* Toggle between hardware and software scroll. */

  cons_org0();
  softscroll = !softscroll;
  printf("%sware scrolling enabled.\n", softscroll ? "Soft" : "Hard");
}


/*===========================================================================*
 *                              cons_stop                                    *
 *===========================================================================*/
PUBLIC void cons_stop()
{
/* Prepare for halt or reboot. */

  cons_org0();
  softscroll = 1;
  select_console(0);
  cons_table[0].c_attr = cons_table[0].c_blank = BLANK_COLOR;
}


/*===========================================================================*
 *                              cons_org0                                    *
 *===========================================================================*/
PRIVATE void cons_org0()
{
/* Scroll video memory back to put the origin at 0. */

  int cons_line;
  console_t *cons;
  unsigned n;

  for (cons_line = 0; cons_line < nr_cons; cons_line++) {
        cons = &cons_table[cons_line];
        while (cons->c_org > cons->c_start) {
                n = vid_size - scr_size;        /* amount of unused memory */
                if (n > cons->c_org - cons->c_start)
                        n = cons->c_org - cons->c_start;
                vid_vid_copy(cons->c_org, cons->c_org - n, scr_size);
                cons->c_org -= n;
        }
        flush(cons);
  }
  select_console(current);
}


/*===========================================================================*
 *                              select_console                               *
 *===========================================================================*/
PUBLIC void select_console(int cons_line)
{
/* Set the current console to console number 'cons_line'. */
```

```
  if (cons_line < 0 || cons_line >= nr_cons) return;
  current = cons_line;
  curcons = &cons_table[cons_line];
  set_6845(VID_ORG, curcons->c_org);
  set_6845(CURSOR, curcons->c_cur);
}


/*===========================================================================*
 *                              con_loadfont                                 *
 *===========================================================================*/
PUBLIC int con_loadfont(user_phys)
phys_bytes user_phys;
{
/* Load a font into the EGA or VGA adapter. */

  static struct sequence seq1[7] = {
        { GA_SEQUENCER_INDEX, 0x00, 0x01 },
        { GA_SEQUENCER_INDEX, 0x02, 0x04 },
        { GA_SEQUENCER_INDEX, 0x04, 0x07 },
        { GA_SEQUENCER_INDEX, 0x00, 0x03 },
        { GA_GRAPHICS_INDEX, 0x04, 0x02 },
        { GA_GRAPHICS_INDEX, 0x05, 0x00 },
        { GA_GRAPHICS_INDEX, 0x06, 0x00 },
  };
  static struct sequence seq2[7] = {
        { GA_SEQUENCER_INDEX, 0x00, 0x01 },
        { GA_SEQUENCER_INDEX, 0x02, 0x03 },
        { GA_SEQUENCER_INDEX, 0x04, 0x03 },
        { GA_SEQUENCER_INDEX, 0x00, 0x03 },
        { GA_GRAPHICS_INDEX, 0x04, 0x00 },
        { GA_GRAPHICS_INDEX, 0x05, 0x10 },
        { GA_GRAPHICS_INDEX, 0x06,    0 },
  };

  seq2[6].value= color ? 0x0E : 0x0A;

  if (!ega) return(ENOTTY);

  lock();
  ga_program(seq1);     /* bring font memory into view */

  phys_copy(user_phys, (phys_bytes)GA_VIDEO_ADDRESS, (phys_bytes)GA_FONT_SIZE);

  ga_program(seq2);     /* restore */
  unlock();

  return(OK);
}


/*===========================================================================*
 *                              ga_program                                   *
 *===========================================================================*/
PRIVATE void ga_program(seq)
struct sequence *seq;
{
/* support function for con_loadfont */

```

```
  int len= 7;
  do {
        out_byte(seq->index, seq->port);
        out_byte(seq->index+1, seq->value);
        seq++;
  } while (--len > 0);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/kernel/dmp.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains some dumping routines for debugging. */

#include "kernel.h"
#include <minix/com.h>
#include "proc.h"

char *vargv;

FORWARD _PROTOTYPE(char *proc_name, (int proc_nr));

/*===========================================================================*
 *                              p_dmp                                        *
 *===========================================================================*/
PUBLIC void p_dmp()
{
/* Proc table dump */

  register struct proc *rp;
  static struct proc *oldrp = BEG_PROC_ADDR;
  int n = 0;
  phys_clicks text, data, size;
  int proc_nr;

  printf("\n--pid --pc- ---sp- flag -user --sys-- -text- -data- -size- -recv- command\n");

  for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        proc_nr = proc_number(rp);
        if (rp->p_flags & P_SLOT_FREE) continue;
        if (++n > 20) break;
        text = rp->p_map[T].mem_phys;
        data = rp->p_map[D].mem_phys;
        size = rp->p_map[T].mem_len
                + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len) - data);
        printf("%5d %5lx %6lx %2x %7U %7U %5uK %5uK %5uK ",
               proc_nr < 0 ? proc_nr : rp->p_pid,
               (unsigned long) rp->p_reg.pc,
               (unsigned long) rp->p_reg.sp,
               rp->p_flags,
               rp->user_time, rp->sys_time,
               click_to_round_k(text), click_to_round_k(data),
               click_to_round_k(size));
        if (rp->p_flags & RECEIVING) {
                printf("%-7.7s", proc_name(rp->p_getfrom));
        } else
        if (rp->p_flags & SENDING) {
```

```
                printf("S:%-5.5s", proc_name(rp->p_sendto));
        } else
        if (rp->p_flags == 0) {
                printf("       ");
        }
        printf("%s\n", rp->p_name);
  }
  if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
  oldrp = rp;
}


/*===========================================================================*
 *                              map_dmp                                      *
 *===========================================================================*/
PUBLIC void map_dmp()
{
  register struct proc *rp;
  static struct proc *oldrp = cproc_addr(HARDWARE);
  int n = 0;
  phys_clicks size;

  printf("\nPROC NAME-  -----TEXT-----  -----DATA-----  ----STACK-----  -SIZE-\n");
  for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        if (rp->p_flags & P_SLOT_FREE) continue;
        if (++n > 20) break;
        size = rp->p_map[T].mem_len
                + ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len)
                                                - rp->p_map[D].mem_phys);
        printf("%3d %-6.6s  %4x %4x %4x  %4x %4x %4x  %4x %4x %4x  %5uK\n",
               proc_number(rp),
               rp->p_name,
               rp->p_map[T].mem_vir, rp->p_map[T].mem_phys, rp->p_map[T].mem_len,
               rp->p_map[D].mem_vir, rp->p_map[D].mem_phys, rp->p_map[D].mem_len,
               rp->p_map[S].mem_vir, rp->p_map[S].mem_phys, rp->p_map[S].mem_len,
               click_to_round_k(size));
  }
  if (rp == END_PROC_ADDR) rp = cproc_addr(HARDWARE); else printf("--more--\r");
  oldrp = rp;
}


/*===========================================================================*
 *                              proc_name                                    *
 *===========================================================================*/
PRIVATE char *proc_name(proc_nr)
int proc_nr;
{
  if (proc_nr == ANY) return "ANY";
  return proc_addr(proc_nr)->p_name;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/kernel/system.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This task handles the interface between file system and kernel as well as
 * between memory manager and kernel.  System services are obtained by sending
 * sys_task() a message specifying what is needed.  To make life easier for
 * MM and FS, a library is provided with routines whose names are of the
 * form sys_xxx, e.g. sys_xit sends the SYS_XIT message to sys_task.  The
 * message types and parameters are:
 *
 *   SYS_FORK    informs kernel that a process has forked
 *   SYS_NEWMAP  allows MM to set up a process memory map
 *   SYS_GETMAP  allows MM to get a process' memory map
 *   SYS_EXEC    sets program counter and stack pointer after EXEC
 *   SYS_XIT     informs kernel that a process has exited
 *   SYS_GETSP   caller wants to read out some process' stack pointer
 *   SYS_TIMES   caller wants to get accounting times for a process
 *   SYS_ABORT   MM or FS cannot go on; abort MINIX
 *   SYS_FRESH   start with a fresh process image during EXEC (68000 only)
 *   SYS_SENDSIG send a signal to a process (POSIX style)
 *   SYS_SIGRETURN complete POSIX-style signalling
 *   SYS_KILL    cause a signal to be sent via MM
 *   SYS_ENDSIG  finish up after SYS_KILL-type signal
 *   SYS_COPY    request a block of data to be copied between processes
 *   SYS_VCOPY   request a series of data blocks to be copied between procs
 *   SYS_GBOOT   copies the boot parameters to a process
 *   SYS_MEM     returns the next free chunk of physical memory
 *   SYS_UMAP    compute the physical address for a given virtual address
 *   SYS_TRACE   request a trace operation
 *
 * Message types and parameters:
 *
 *    m_type       PROC1     PROC2      PID     MEM_PTR
 * ------------------------------------------------------
 * | SYS_FORK    | parent  |  child  |   pid   |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_NEWMAP  | proc nr |         |         | map ptr |
 * |------------+---------+---------+---------+---------|
 * | SYS_EXEC    | proc nr | traced  | new sp  |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_XIT     | parent  | exitee  |         |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_GETSP   | proc nr |         |         |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_TIMES   | proc nr |         | buf ptr |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_ABORT   |         |         |         |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_FRESH   | proc nr | data_cl |         |         |
 * |------------+---------+---------+---------+---------|
 * | SYS_GBOOT   | proc nr |         |         | bootptr |
 * |------------+---------+---------+---------+---------|
 * | SYS_GETMAP  | proc nr |         |         | map ptr |
 * ------------------------------------------------------
 *
 *    m_type          m1_i1     m1_i2     m1_i3       m1_p1
 * ----------------+---------+---------+---------+--------------
 * | SYS_VCOPY      |  src p  |  dst p  | vec siz | vc addr     |
```

```
 * |--------------+---------+---------+---------+-------------|
 * | SYS_SENDSIG  | proc nr |         |         | smp         |
 * |--------------+---------+---------+---------+-------------|
 * | SYS_SIGRETURN | proc nr |        |         | scp         |
 * |--------------+---------+---------+---------+-------------|
 * | SYS_ENDSIG   | proc nr |         |         |             |
 * -------------------------------------------------------------
 *
 *    m_type       m2_i1     m2_i2     m2_l1     m2_l2
 * ------------------------------------------------------
 * | SYS_TRACE  | proc_nr | request |  addr   |  data   |
 * ------------------------------------------------------
 *
 *
 *    m_type       m6_i1     m6_i2     m6_i3     m6_f1
 * ------------------------------------------------------
 * | SYS_KILL   | proc_nr  |  sig    |         |         |
 * ------------------------------------------------------
 *
 *
 *    m_type      m5_c1   m5_i1    m5_l1   m5_c2   m5_i2    m5_l2   m5_l3
 * --------------------------------------------------------------------------
 * | SYS_COPY   |src seg|src proc|src vir|dst seg|dst proc|dst vir| byte ct |
 * --------------------------------------------------------------------------
 * | SYS_UMAP   |  seg  |proc nr |vir adr|       |        |       | byte ct |
 * --------------------------------------------------------------------------
 *
 *
 *    m_type      m1_i1      m1_i2      m1_i3
 * |------------+----------+----------+----------
 * | SYS_MEM    | mem base | mem size | tot mem |
 * ----------------------------------------------
 *
 * In addition to the main sys_task() entry point, there are 5 other minor
 * entry points:
 *   cause_sig: take action to cause a signal to occur, sooner or later
 *   inform:    tell MM about pending signals
 *   numap:     umap D segment starting from process number instead of pointer
 *   umap:      compute the physical address for a given virtual address
 *   alloc_segments: allocate segments for 8088 or higher processor
 */

#include "kernel.h"
#include <signal.h>
#include <unistd.h>
#include <sys/sigcontext.h>
#include <sys/ptrace.h>
#include <minix/boot.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "proc.h"
#include "protect.h"

/* PSW masks. */
#define IF_MASK 0x00000200
#define IOPL_MASK 0x003000

PRIVATE message m;

FORWARD _PROTOTYPE( int do_abort, (message *m_ptr) );
```

```
FORWARD _PROTOTYPE( int do_copy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_exec, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_fork, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_gboot, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getsp, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_kill, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_mem, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_newmap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_sendsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_sigreturn, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_endsig, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_times, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_trace, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_umap, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_xit, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_vcopy, (message *m_ptr) );
FORWARD _PROTOTYPE( int do_getmap, (message *m_ptr) );


/*===========================================================================*
 *                              sys_task                                     *
 *===========================================================================*/
PUBLIC void sys_task()
{
/* Main entry point of sys_task.  Get the message and dispatch on type. */

  register int r;

  while (TRUE) {
        receive(ANY, &m);

        switch (m.m_type) {     /* which system call */
            case SYS_FORK:      r = do_fork(&m);        break;
            case SYS_NEWMAP:    r = do_newmap(&m);      break;
            case SYS_GETMAP:    r = do_getmap(&m);      break;
            case SYS_EXEC:      r = do_exec(&m);        break;
            case SYS_XIT:       r = do_xit(&m);         break;
            case SYS_GETSP:     r = do_getsp(&m);       break;
            case SYS_TIMES:     r = do_times(&m);       break;
            case SYS_ABORT:     r = do_abort(&m);       break;
            case SYS_SENDSIG:   r = do_sendsig(&m);     break;
            case SYS_SIGRETURN: r = do_sigreturn(&m);   break;
            case SYS_KILL:      r = do_kill(&m);        break;
            case SYS_ENDSIG:    r = do_endsig(&m);      break;
            case SYS_COPY:      r = do_copy(&m);        break;
            case SYS_VCOPY:     r = do_vcopy(&m);       break;
            case SYS_GBOOT:     r = do_gboot(&m);       break;
            case SYS_MEM:       r = do_mem(&m);         break;
            case SYS_UMAP:      r = do_umap(&m);        break;
            case SYS_TRACE:     r = do_trace(&m);       break;
            default:            r = E_BAD_FCN;
        }

        m.m_type = r;           /* 'r' reports status of call */
        send(m.m_source, &m);   /* send reply to caller */
  }
}


/*===========================================================================*
```

```
 *                              do_fork                                    *
 *===========================================================================*/
PRIVATE int do_fork(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_fork().  m_ptr->PROC1 has forked.  The child is m_ptr->PROC2. */

  reg_t old_ldt_sel;
  register struct proc *rpc;
  struct proc *rpp;

  if (!isoksusern(m_ptr->PROC1) || !isoksusern(m_ptr->PROC2))
        return(E_BAD_PROC);
  rpp = proc_addr(m_ptr->PROC1);
  rpc = proc_addr(m_ptr->PROC2);

  /* Copy parent 'proc' struct to child. */
  old_ldt_sel = rpc->p_ldt_sel; /* stop this being obliterated by copy */

  *rpc = *rpp;                  /* copy 'proc' struct */

  rpc->p_ldt_sel = old_ldt_sel;
  rpc->p_nr = m_ptr->PROC2;     /* this was obliterated by copy */

  rpc->p_flags |= NO_MAP;       /* inhibit the process from running */

  rpc->p_flags &= ~(PENDING | SIG_PENDING | P_STOP);

  /* Only 1 in group should have PENDING, child does not inherit trace status*/
  sigemptyset(&rpc->p_pending);
  rpc->p_pendcount = 0;
  rpc->p_pid = m_ptr->PID;      /* install child's pid */
  rpc->p_reg.retreg = 0;        /* child sees pid = 0 to know it is child */

  rpc->user_time = 0;           /* set all the accounting times to 0 */
  rpc->sys_time = 0;
  rpc->child_utime = 0;
  rpc->child_stime = 0;

  return(OK);
}


/*===========================================================================*
 *                              do_newmap                                  *
 *===========================================================================*/
PRIVATE int do_newmap(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_newmap().  Fetch the memory map from MM. */

  register struct proc *rp;
  phys_bytes src_phys;
  int caller;                   /* whose space has the new map (usually MM) */
  int k;                        /* process whose map is to be loaded */
  int old_flags;                /* value of flags before modification */
  struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */

  /* Extract message parameters and copy new memory map from MM. */
  caller = m_ptr->m_source;
```

```
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
  if (!isokprocn(k)) return(E_BAD_PROC);
  rp = proc_addr(k);            /* ptr to entry of user getting new map */

  /* Copy the map from MM. */
  src_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
  if (src_phys == 0) panic("bad call to sys_newmap", NO_NUM);
  phys_copy(src_phys, vir2phys(rp->p_map), (phys_bytes) sizeof(rp->p_map));

  alloc_segments(rp);
  old_flags = rp->p_flags;      /* save the previous value of the flags */
  rp->p_flags &= ~NO_MAP;
  if (old_flags != 0 && rp->p_flags == 0) lock_ready(rp);

  return(OK);
}


/*===========================================================================*
 *                              do_getmap                                    *
 *===========================================================================*/
PRIVATE int do_getmap(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_getmap().  Report the memory map to MM. */

  register struct proc *rp;
  phys_bytes dst_phys;
  int caller;                   /* where the map has to be stored */
  int k;                        /* process whose map is to be loaded */
  struct mem_map *map_ptr;      /* virtual address of map inside caller (MM) */

  /* Extract message parameters and copy new memory map to MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;

  if (!isokprocn(k))
        panic("do_getmap got bad proc: ", m_ptr->PROC1);

  rp = proc_addr(k);            /* ptr to entry of the map */

  /* Copy the map to MM. */
  dst_phys = umap(proc_addr(caller), D, (vir_bytes) map_ptr, sizeof(rp->p_map));
  if (dst_phys == 0) panic("bad call to sys_getmap", NO_NUM);
  phys_copy(vir2phys(rp->p_map), dst_phys, sizeof(rp->p_map));

  return(OK);
}


/*===========================================================================*
 *                              do_exec                                      *
 *===========================================================================*/
PRIVATE int do_exec(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_exec().  A process has done a successful EXEC. Patch it up. */

```

```
  register struct proc *rp;
  reg_t sp;                     /* new sp */
  phys_bytes phys_name;
  char *np;
#define NLEN (sizeof(rp->p_name)-1)

  if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
  /* PROC2 field is used as flag to indicate process is being traced */
  if (m_ptr->PROC2) cause_sig(m_ptr->PROC1, SIGTRAP);
  sp = (reg_t) m_ptr->STACK_PTR;
  rp = proc_addr(m_ptr->PROC1);
  rp->p_reg.sp = sp;            /* set the stack pointer */
  rp->p_reg.pc = (reg_t) m_ptr->IP_PTR; /* set pc */
  rp->p_alarm = 0;              /* reset alarm timer */
  rp->p_flags &= ˜RECEIVING;    /* MM does not reply to EXEC call */
  if (rp->p_flags == 0) lock_ready(rp);

  /* Save command name for debugging, ps(1) output, etc. */
  phys_name = numap(m_ptr->m_source, (vir_bytes) m_ptr->NAME_PTR,
                                                      (vir_bytes) NLEN);
  if (phys_name != 0) {
        phys_copy(phys_name, vir2phys(rp->p_name), (phys_bytes) NLEN);
        for (np = rp->p_name; (*np & BYTE) >= ' '; np++) {}
        *np = 0;
  }
  return(OK);
}


/*===========================================================================*
 *                              do_xit                                       *
 *===========================================================================*/
PRIVATE int do_xit(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_xit().  A process has exited. */

  register struct proc *rp, *rc;
  struct proc *np, *xp;
  int parent;                   /* number of exiting proc's parent */
  int proc_nr;                  /* number of process doing the exit */
  phys_clicks base, size;

  parent = m_ptr->PROC1;        /* slot number of parent process */
  proc_nr = m_ptr->PROC2;       /* slot number of exiting process */
  if (!isoksusern(parent) || !isoksusern(proc_nr)) return(E_BAD_PROC);
  rp = proc_addr(parent);
  rc = proc_addr(proc_nr);
  lock();
  rp->child_utime += rc->user_time + rc->child_utime;   /* accum child times */
  rp->child_stime += rc->sys_time + rc->child_stime;
  unlock();
  rc->p_alarm = 0;              /* turn off alarm timer */
  if (rc->p_flags == 0) lock_unready(rc);

  strcpy(rc->p_name, "<noname>");       /* process no longer has a name */

  /* If the process being terminated happens to be queued trying to send a
   * message (i.e., the process was killed by a signal, rather than it doing an
   * EXIT), then it must be removed from the message queues.
```

```
   */
  if (rc->p_flags & SENDING) {
        /* Check all proc slots to see if the exiting process is queued. */
        for (rp = BEG_PROC_ADDR; rp < END_PROC_ADDR; rp++) {
                if (rp->p_callerq == NIL_PROC) continue;
                if (rp->p_callerq == rc) {
                        /* Exiting process is on front of this queue. */
                        rp->p_callerq = rc->p_sendlink;
                        break;
                } else {
                        /* See if exiting process is in middle of queue. */
                        np = rp->p_callerq;
                        while ( ( xp = np->p_sendlink) != NIL_PROC)
                                if (xp == rc) {
                                        np->p_sendlink = xp->p_sendlink;
                                        break;
                                } else {
                                        np = xp;
                                }
                }
        }
  }

  if (rc->p_flags & PENDING) --sig_procs;
  sigemptyset(&rc->p_pending);
  rc->p_pendcount = 0;
  rc->p_flags = P_SLOT_FREE;
  return(OK);
}


/*===========================================================================*
 *                              do_getsp                                     *
 *===========================================================================*/
PRIVATE int do_getsp(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_getsp().  MM wants to know what sp is. */

  register struct proc *rp;

  if (!isoksusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);
  m_ptr->STACK_PTR = (char *) rp->p_reg.sp;     /* return sp here (bad type) */
  return(OK);
}


/*===========================================================================*
 *                              do_times                                     *
 *===========================================================================*/
PRIVATE int do_times(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_times().  Retrieve the accounting information. */

  register struct proc *rp;

  if (!isoksusern(m_ptr->PROC1)) return E_BAD_PROC;
  rp = proc_addr(m_ptr->PROC1);
```

```

  /* Insert the times needed by the TIMES system call in the message. */
  lock();                       /* halt the volatile time counters in rp */
  m_ptr->USER_TIME   = rp->user_time;
  m_ptr->SYSTEM_TIME = rp->sys_time;
  unlock();
  m_ptr->CHILD_UTIME = rp->child_utime;
  m_ptr->CHILD_STIME = rp->child_stime;
  m_ptr->BOOT_TICKS  = get_uptime();
  return(OK);
}


/*===========================================================================*
 *                              do_abort                                     *
 *===========================================================================*/
PRIVATE int do_abort(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_abort.  MINIX is unable to continue.  Terminate operation. */
  char monitor_code[64];
  phys_bytes src_phys;

  if (m_ptr->m1_i1 == RBT_MONITOR) {
        /* The monitor is to run user specified instructions. */
        src_phys = numap(m_ptr->m_source, (vir_bytes) m_ptr->m1_p1,
                                        (vir_bytes) sizeof(monitor_code));
        if (src_phys == 0) panic("bad monitor code from", m_ptr->m_source);
        phys_copy(src_phys, vir2phys(monitor_code),
                                        (phys_bytes) sizeof(monitor_code));
        reboot_code = vir2phys(monitor_code);
  }
  wreboot(m_ptr->m1_i1);
  return(OK);                   /* pro-forma (really EDISASTER) */
}




/*===========================================================================*
 *                            do_sendsig                                     *
 *===========================================================================*/
PRIVATE int do_sendsig(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Handle sys_sendsig, POSIX-style signal */

  struct sigmsg smsg;
  register struct proc *rp;
  phys_bytes src_phys, dst_phys;
  struct sigcontext sc, *scp;
  struct sigframe fr, *frp;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

  /* Get the sigmsg structure into our address space.  */
  src_phys = umap(proc_addr(MM_PROC_NR), D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
                  (vir_bytes) sizeof(struct sigmsg));
  if (src_phys == 0)
```

```
        panic("do_sendsig can't signal: bad sigmsg address from MM", NO_NUM);
  phys_copy(src_phys, vir2phys(&smsg), (phys_bytes) sizeof(struct sigmsg));

  /* Compute the usr stack pointer value where sigcontext will be stored. */
  scp = (struct sigcontext *) smsg.sm_stkptr - 1;

  /* Copy the registers to the sigcontext structure. */
  memcpy(&sc.sc_regs, &rp->p_reg, sizeof(struct sigregs));

  /* Finish the sigcontext initialization. */
  sc.sc_flags = SC_SIGCONTEXT;

  sc.sc_mask = smsg.sm_mask;

  /* Copy the sigcontext structure to the user's stack. */
  dst_phys = umap(rp, D, (vir_bytes) scp,
                  (vir_bytes) sizeof(struct sigcontext));
  if (dst_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&sc), dst_phys, (phys_bytes) sizeof(struct sigcontext));

  /* Initialize the sigframe structure. */
  frp = (struct sigframe *) scp - 1;
  fr.sf_scpcopy = scp;
  fr.sf_retadr2= (void (*)()) rp->p_reg.pc;
  fr.sf_fp = rp->p_reg.fp;
  rp->p_reg.fp = (reg_t) &frp->sf_fp;
  fr.sf_scp = scp;
  fr.sf_code = 0;       /* XXX - should be used for type of FP exception */
  fr.sf_signo = smsg.sm_signo;
  fr.sf_retadr = (void (*)()) smsg.sm_sigreturn;

  /* Copy the sigframe structure to the user's stack. */
  dst_phys = umap(rp, D, (vir_bytes) frp, (vir_bytes) sizeof(struct sigframe));
  if (dst_phys == 0) return(EFAULT);
  phys_copy(vir2phys(&fr), dst_phys, (phys_bytes) sizeof(struct sigframe));

  /* Reset user registers to execute the signal handler. */
  rp->p_reg.sp = (reg_t) frp;
  rp->p_reg.pc = (reg_t) smsg.sm_sighandler;

  return(OK);
}

/*===========================================================================*
 *                              do_sigreturn                                 *
 *===========================================================================*/
PRIVATE int do_sigreturn(m_ptr)
register message *m_ptr;
{
/* POSIX style signals require sys_sigreturn to put things in order before the
 * signalled process can resume execution
 */

  struct sigcontext sc;
  register struct proc *rp;
  phys_bytes src_phys;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

```

```
  /* Copy in the sigcontext structure. */
  src_phys = umap(rp, D, (vir_bytes) m_ptr->SIG_CTXT_PTR,
                  (vir_bytes) sizeof(struct sigcontext));
  if (src_phys == 0) return(EFAULT);
  phys_copy(src_phys, vir2phys(&sc), (phys_bytes) sizeof(struct sigcontext));

  /* Make sure that this is not just a jmp_buf. */
  if ((sc.sc_flags & SC_SIGCONTEXT) == 0) return(EINVAL);

  /* Fix up only certain key registers if the compiler doesn't use
   * register variables within functions containing setjmp.
   */
  if (sc.sc_flags & SC_NOREGLOCALS) {
        rp->p_reg.retreg = sc.sc_retreg;
        rp->p_reg.fp = sc.sc_fp;
        rp->p_reg.pc = sc.sc_pc;
        rp->p_reg.sp = sc.sc_sp;
        return (OK);
  }
  sc.sc_psw  = rp->p_reg.psw;

#if (CHIP == INTEL)
  /* Don't panic kernel if user gave bad selectors. */
  sc.sc_cs = rp->p_reg.cs;
  sc.sc_ds = rp->p_reg.ds;
  sc.sc_es = rp->p_reg.es;
#if _WORD_SIZE == 4
  sc.sc_fs = rp->p_reg.fs;
  sc.sc_gs = rp->p_reg.gs;
#endif
#endif

  /* Restore the registers. */
  memcpy(&rp->p_reg, (char *)&sc.sc_regs, sizeof(struct sigregs));

  return(OK);
}

/*===========================================================================*
 *                              do_kill                                      *
 *===========================================================================*/
PRIVATE int do_kill(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_kill(). Cause a signal to be sent to a process via MM.
 * Note that this has nothing to do with the kill (2) system call, this
 * is how the FS (and possibly other servers) get access to cause_sig to
 * send a KSIG message to MM
 */

  if (!isokusern(m_ptr->PR)) return(E_BAD_PROC);
  cause_sig(m_ptr->PR, m_ptr->SIGNUM);
  return(OK);
}


/*===========================================================================*
 *                            do_endsig                                      *
 *===========================================================================*/
PRIVATE int do_endsig(m_ptr)
```

```
register message *m_ptr;        /* pointer to request message */
{
/* Finish up after a KSIG-type signal, caused by a SYS_KILL message or a call
 * to cause_sig by a task
 */

  register struct proc *rp;

  if (!isokusern(m_ptr->PROC1)) return(E_BAD_PROC);
  rp = proc_addr(m_ptr->PROC1);

  /* MM has finished one KSIG. */
  if (rp->p_pendcount != 0 && --rp->p_pendcount == 0
      && (rp->p_flags &= ~SIG_PENDING) == 0)
        lock_ready(rp);
  return(OK);
}

/*===========================================================================*
 *                              do_copy                                      *
 *===========================================================================*/
PRIVATE int do_copy(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_copy().  Copy data for MM or FS. */

  int src_proc, dst_proc, src_space, dst_space;
  vir_bytes src_vir, dst_vir;
  phys_bytes src_phys, dst_phys, bytes;

  /* Dismember the command message. */
  src_proc = m_ptr->SRC_PROC_NR;
  dst_proc = m_ptr->DST_PROC_NR;
  src_space = m_ptr->SRC_SPACE;
  dst_space = m_ptr->DST_SPACE;
  src_vir = (vir_bytes) m_ptr->SRC_BUFFER;
  dst_vir = (vir_bytes) m_ptr->DST_BUFFER;
  bytes = (phys_bytes) m_ptr->COPY_BYTES;

  /* Compute the source and destination addresses and do the copy. */
  if (src_proc == ABS)
        src_phys = (phys_bytes) m_ptr->SRC_BUFFER;
  else {
        if (bytes != (vir_bytes) bytes)
                /* This would happen for 64K segments and 16-bit vir_bytes.
                 * It would happen a lot for do_fork except MM uses ABS
                 * copies for that case.
                 */
                panic("overflow in count in do_copy", NO_NUM);

        src_phys = umap(proc_addr(src_proc), src_space, src_vir,
                        (vir_bytes) bytes);
        }

  if (dst_proc == ABS)
        dst_phys = (phys_bytes) m_ptr->DST_BUFFER;
  else
        dst_phys = umap(proc_addr(dst_proc), dst_space, dst_vir,
                        (vir_bytes) bytes);

```

```c
  if (src_phys == 0 || dst_phys == 0) return(EFAULT);
  phys_copy(src_phys, dst_phys, bytes);
  return(OK);
}


/*===========================================================================*
 *                              do_vcopy                                     *
 *===========================================================================*/
PRIVATE int do_vcopy(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Handle sys_vcopy(). Copy multiple blocks of memory */

  int src_proc, dst_proc, vect_s, i;
  vir_bytes src_vir, dst_vir, vect_addr;
  phys_bytes src_phys, dst_phys, bytes;
  cpvec_t cpvec_table[CPVEC_NR];

  /* Dismember the command message. */
  src_proc = m_ptr->m1_i1;
  dst_proc = m_ptr->m1_i2;
  vect_s = m_ptr->m1_i3;
  vect_addr = (vir_bytes)m_ptr->m1_p1;

  if (vect_s > CPVEC_NR) return EDOM;

  src_phys= numap (m_ptr->m_source, vect_addr, vect_s * sizeof(cpvec_t));
  if (!src_phys) return EFAULT;
  phys_copy(src_phys, vir2phys(cpvec_table),
                                (phys_bytes) (vect_s * sizeof(cpvec_t)));

  for (i = 0; i < vect_s; i++) {
        src_vir= cpvec_table[i].cpv_src;
        dst_vir= cpvec_table[i].cpv_dst;
        bytes= cpvec_table[i].cpv_size;
        src_phys = numap(src_proc,src_vir,(vir_bytes)bytes);
        dst_phys = numap(dst_proc,dst_vir,(vir_bytes)bytes);
        if (src_phys == 0 || dst_phys == 0) return(EFAULT);
        phys_copy(src_phys, dst_phys, bytes);
  }
  return(OK);
}


/*==========================================================================*
 *                              do_gboot                                    *
 *==========================================================================*/
PUBLIC struct bparam_s boot_parameters;

PRIVATE int do_gboot(m_ptr)
message *m_ptr;                 /* pointer to request message */
{
/* Copy the boot parameters.  Normally only called during fs init. */

  phys_bytes dst_phys;

  dst_phys = umap(proc_addr(m_ptr->PROC1), D, (vir_bytes) m_ptr->MEM_PTR,
                                (vir_bytes) sizeof(boot_parameters));
  if (dst_phys == 0) panic("bad call to SYS_GBOOT", NO_NUM);
```

```
  phys_copy(vir2phys(&boot_parameters), dst_phys,
                              (phys_bytes) sizeof(boot_parameters));
  return(OK);
}


/*===========================================================================*
 *                              do_mem                                       *
 *===========================================================================*/
PRIVATE int do_mem(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Return the base and size of the next chunk of memory. */

  struct memory *memp;

  for (memp = mem; memp < &mem[NR_MEMS]; ++memp) {
        m_ptr->m1_i1 = memp->base;
        m_ptr->m1_i2 = memp->size;
        m_ptr->m1_i3 = tot_mem_size;
        memp->size = 0;
        if (m_ptr->m1_i2 != 0) break;           /* found a chunk */
  }
  return(OK);
}


/*==========================================================================*
 *                              do_umap                                     *
 *==========================================================================*/
PRIVATE int do_umap(m_ptr)
register message *m_ptr;        /* pointer to request message */
{
/* Same as umap(), for non-kernel processes. */

  m_ptr->SRC_BUFFER = umap(proc_addr((int) m_ptr->SRC_PROC_NR),
                          (int) m_ptr->SRC_SPACE,
                          (vir_bytes) m_ptr->SRC_BUFFER,
                          (vir_bytes) m_ptr->COPY_BYTES);
  return(OK);
}


/*==========================================================================*
 *                              do_trace                                    *
 *==========================================================================*/
#define TR_PROCNR       (m_ptr->m2_i1)
#define TR_REQUEST      (m_ptr->m2_i2)
#define TR_ADDR         ((vir_bytes) m_ptr->m2_l1)
#define TR_DATA         (m_ptr->m2_l2)
#define TR_VLSIZE       ((vir_bytes) sizeof(long))

PRIVATE int do_trace(m_ptr)
register message *m_ptr;
{
/* Handle the debugging commands supported by the ptrace system call
 * The commands are:
 * T_STOP       stop the process
 * T_OK         enable tracing by parent for this process
 * T_GETINS     return value from instruction space
```

```
 * T_GETDATA    return value from data space
 * T_GETUSER    return value from user process table
 * T_SETINS     set value from instruction space
 * T_SETDATA    set value from data space
 * T_SETUSER    set value in user process table
 * T_RESUME     resume execution
 * T_EXIT       exit
 * T_STEP       set trace bit
 *
 * The T_OK and T_EXIT commands are handled completely by the memory manager,
 * all others come here.
 */

  register struct proc *rp;
  phys_bytes src, dst;
  int i;

  rp = proc_addr(TR_PROCNR);
  if (rp->p_flags & P_SLOT_FREE) return(EIO);
  switch (TR_REQUEST) {
  case T_STOP:                  /* stop process */
        if (rp->p_flags == 0) lock_unready(rp);
        rp->p_flags |= P_STOP;
        rp->p_reg.psw &= ~TRACEBIT;     /* clear trace bit */
        return(OK);

  case T_GETINS:                /* return value from instruction space */
        if (rp->p_map[T].mem_len != 0) {
                if ((src = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
                phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
                break;
        }
        /* Text space is actually data space - fall through. */

  case T_GETDATA:               /* return value from data space */
        if ((src = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
        phys_copy(src, vir2phys(&TR_DATA), (phys_bytes) sizeof(long));
        break;

  case T_GETUSER:               /* return value from process table */
        if ((TR_ADDR & (sizeof(long) - 1)) != 0 ||
            TR_ADDR > sizeof(struct proc) - sizeof(long))
                return(EIO);
        TR_DATA = *(long *) ((char *) rp + (int) TR_ADDR);
        break;

  case T_SETINS:                /* set value in instruction space */
        if (rp->p_map[T].mem_len != 0) {
                if ((dst = umap(rp, T, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
                phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
                TR_DATA = 0;
                break;
        }
        /* Text space is actually data space - fall through. */

  case T_SETDATA:                       /* set value in data space */
        if ((dst = umap(rp, D, TR_ADDR, TR_VLSIZE)) == 0) return(EIO);
        phys_copy(vir2phys(&TR_DATA), dst, (phys_bytes) sizeof(long));
        TR_DATA = 0;
        break;
```

```c

  case T_SETUSER:                       /* set value in process table */
        if ((TR_ADDR & (sizeof(reg_t) - 1)) != 0 ||
             TR_ADDR > sizeof(struct stackframe_s) - sizeof(reg_t))
                return(EIO);
        i = (int) TR_ADDR;
#if (CHIP == INTEL)
        /* Altering segment registers might crash the kernel when it
         * tries to load them prior to restarting a process, so do
         * not allow it.
         */
        if (i == (int) &((struct proc *) 0)->p_reg.cs ||
            i == (int) &((struct proc *) 0)->p_reg.ds ||
            i == (int) &((struct proc *) 0)->p_reg.es ||
#if _WORD_SIZE == 4
            i == (int) &((struct proc *) 0)->p_reg.gs ||
            i == (int) &((struct proc *) 0)->p_reg.fs ||
#endif
            i == (int) &((struct proc *) 0)->p_reg.ss)
                return(EIO);
#endif
        if (i == (int) &((struct proc *) 0)->p_reg.psw)
                /* only selected bits are changeable */
                SETPSW(rp, TR_DATA);
        else
                *(reg_t *) ((char *) &rp->p_reg + i) = (reg_t) TR_DATA;
        TR_DATA = 0;
        break;

  case T_RESUME:                /* resume execution */
        rp->p_flags &= ~P_STOP;
        if (rp->p_flags == 0) lock_ready(rp);
        TR_DATA = 0;
        break;

  case T_STEP:                  /* set trace bit */
        rp->p_reg.psw |= TRACEBIT;
        rp->p_flags &= ~P_STOP;
        if (rp->p_flags == 0) lock_ready(rp);
        TR_DATA = 0;
        break;

  default:
        return(EIO);
  }
  return(OK);
}

/*===========================================================================*
 *                              cause_sig                                    *
 *===========================================================================*/
PUBLIC void cause_sig(proc_nr, sig_nr)
int proc_nr;                    /* process to be signalled */
int sig_nr;                     /* signal to be sent, 1 to _NSIG */
{
/* A task wants to send a signal to a process.   Examples of such tasks are:
 *   TTY wanting to cause SIGINT upon getting a DEL
 *   CLOCK wanting to cause SIGALRM when timer expires
 * FS also uses this to send a signal, via the SYS_KILL message.
 * Signals are handled by sending a message to MM.  The tasks don't dare do
```

```
 * that directly, for fear of what would happen if MM were busy.  Instead they
 * call cause_sig, which sets bits in p_pending, and then carefully checks to
 * see if MM is free.  If so, a message is sent to it.  If not, when it becomes
 * free, a message is sent.  The process being signaled is blocked while MM
 * has not seen or finished with all signals for it.  These signals are
 * counted in p_pendcount, and the SIG_PENDING flag is kept nonzero while
 * there are some.  It is not sufficient to ready the process when MM is
 * informed, because MM can block waiting for FS to do a core dump.
 */

  register struct proc *rp, *mmp;

  rp = proc_addr(proc_nr);
  if (sigismember(&rp->p_pending, sig_nr))
        return;                 /* this signal already pending */
  sigaddset(&rp->p_pending, sig_nr);
  ++rp->p_pendcount;            /* count new signal pending */
  if (rp->p_flags & PENDING)
        return;                 /* another signal already pending */
  if (rp->p_flags == 0) lock_unready(rp);
  rp->p_flags |= PENDING | SIG_PENDING;
  ++sig_procs;                  /* count new process pending */

  mmp = proc_addr(MM_PROC_NR);
  if ( ((mmp->p_flags & RECEIVING) == 0) || mmp->p_getfrom != ANY) return;
  inform();
}


/*===========================================================================*
 *                              inform                                       *
 *===========================================================================*/
PUBLIC void inform()
{
/* When a signal is detected by the kernel (e.g., DEL), or generated by a task
 * (e.g. clock task for SIGALRM), cause_sig() is called to set a bit in the
 * p_pending field of the process to signal.  Then inform() is called to see
 * if MM is idle and can be told about it.  Whenever MM blocks, a check is
 * made to see if 'sig_procs' is nonzero; if so, inform() is called.
 */

  register struct proc *rp;

  /* MM is waiting for new input.  Find a process with pending signals. */
  for (rp = BEG_SERV_ADDR; rp < END_PROC_ADDR; rp++)
        if (rp->p_flags & PENDING) {
                m.m_type = KSIG;
                m.SIG_PROC = proc_number(rp);
                m.SIG_MAP = rp->p_pending;
                sig_procs--;
                if (lock_mini_send(proc_addr(HARDWARE), MM_PROC_NR, &m) != OK)
                        panic("can't inform MM", NO_NUM);
                sigemptyset(&rp->p_pending); /* the ball is now in MM's court */
                rp->p_flags &= ~PENDING;/* remains inhibited by SIG_PENDING */
                lock_pick_proc();       /* avoid delay in scheduling MM */
                return;
        }
}
```

```
/*===========================================================================*
 *                              umap                                         *
 *===========================================================================*/
PUBLIC phys_bytes umap(rp, seg, vir_addr, bytes)
register struct proc *rp;       /* pointer to proc table entry for process */
int seg;                        /* T, D, or S segment */
vir_bytes vir_addr;             /* virtual address in bytes within the seg */
vir_bytes bytes;                /* # of bytes to be copied */
{
/* Calculate the physical memory address for a given virtual address. */

  vir_clicks vc;                /* the virtual address in clicks */
  phys_bytes pa;                /* intermediate variables as phys_bytes */
  phys_bytes seg_base;

  /* If 'seg' is D it could really be S and vice versa.  T really means T.
   * If the virtual address falls in the gap,  it causes a problem. On the
   * 8088 it is probably a legal stack reference, since "stackfaults" are
   * not detected by the hardware.  On 8088s, the gap is called S and
   * accepted, but on other machines it is called D and rejected.
   * The Atari ST behaves like the 8088 in this respect.
   */

  if (bytes <= 0) return( (phys_bytes) 0);
  vc = (vir_addr + bytes - 1) >> CLICK_SHIFT;   /* last click of data */

  if (seg != T)
        seg = (vc < rp->p_map[D].mem_vir + rp->p_map[D].mem_len ? D : S);

  if((vir_addr>>CLICK_SHIFT) >= rp->p_map[seg].mem_vir+ rp->p_map[seg].mem_len)
        return( (phys_bytes) 0 );
  seg_base = (phys_bytes) rp->p_map[seg].mem_phys;
  seg_base = seg_base << CLICK_SHIFT;   /* segment origin in bytes */
  pa = (phys_bytes) vir_addr;
  pa -= rp->p_map[seg].mem_vir << CLICK_SHIFT;
  return(seg_base + pa);
}


/*==========================================================================*
 *                              numap                                       *
 *==========================================================================*/
PUBLIC phys_bytes numap(proc_nr, vir_addr, bytes)
int proc_nr;                    /* process number to be mapped */
vir_bytes vir_addr;             /* virtual address in bytes within D seg */
vir_bytes bytes;                /* # of bytes required in segment  */
{
/* Do umap() starting from a process number instead of a pointer.  This
 * function is used by device drivers, so they need not know about the
 * process table.  To save time, there is no 'seg' parameter. The segment
 * is always D.
 */

  return(umap(proc_addr(proc_nr), D, vir_addr, bytes));
}

#if (CHIP == INTEL)
/*==========================================================================*
 *                              alloc_segments                              *
 *==========================================================================*/
```

```
PUBLIC void alloc_segments(rp)
register struct proc *rp;
{
/* This is called only by do_newmap, but is broken out as a separate function
 * because so much is hardware-dependent.
 */

  phys_bytes code_bytes;
  phys_bytes data_bytes;
  int privilege;

  if (protected_mode) {
	data_bytes = (phys_bytes) (rp->p_map[S].mem_vir + rp->p_map[S].mem_len)
	                << CLICK_SHIFT;
	if (rp->p_map[T].mem_len == 0)
		code_bytes = data_bytes;	/* common I&D, poor protect */
	else
		code_bytes = (phys_bytes) rp->p_map[T].mem_len << CLICK_SHIFT;
	privilege = istaskp(rp) ? TASK_PRIVILEGE : USER_PRIVILEGE;
	init_codeseg(&rp->p_ldt[CS_LDT_INDEX],
		     (phys_bytes) rp->p_map[T].mem_phys << CLICK_SHIFT,
		     code_bytes, privilege);
	init_dataseg(&rp->p_ldt[DS_LDT_INDEX],
		     (phys_bytes) rp->p_map[D].mem_phys << CLICK_SHIFT,
		     data_bytes, privilege);
	rp->p_reg.cs = (CS_LDT_INDEX * DESC_SIZE) | TI | privilege;
#if _WORD_SIZE == 4
	rp->p_reg.gs =
	rp->p_reg.fs =
#endif
	rp->p_reg.ss =
	rp->p_reg.es =
	rp->p_reg.ds = (DS_LDT_INDEX*DESC_SIZE) | TI | privilege;
  } else {
	rp->p_reg.cs = click_to_hclick(rp->p_map[T].mem_phys);
	rp->p_reg.ss =
	rp->p_reg.es =
	rp->p_reg.ds = click_to_hclick(rp->p_map[D].mem_phys);
  }
}
#endif /* (CHIP == INTEL) */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/mm/mm.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This is the master header for mm.  It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1	/* tell headers to include POSIX stuff */
#define _MINIX             1	/* tell headers to include MINIX stuff */
#define _SYSTEM            1	/* tell headers that this is the kernel */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>	/* MUST be first */
#include <ansi.h>		/* MUST be second */
```

```
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>

#include <fcntl.h>
#include <unistd.h>
#include <minix/syslib.h>

#include <limits.h>
#include <errno.h>

#include "const.h"
#include "type.h"
#include "proto.h"
#include "glo.h"
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/mm/const.h
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Constants used by the Memory Manager. */

#define NO_MEM ((phys_clicks) 0)  /* returned by alloc_mem() with mem is up */

#if (CHIP == INTEL && _WORD_SIZE == 2)
/* These definitions are used in size_ok and are not needed for 386.
 * The 386 segment granularity is 1 for segments smaller than 1M and 4096
 * above that.
 */
#define PAGE_SIZE         16    /* how many bytes in a page (s.b.HCLICK_SIZE)*/
#define MAX_PAGES       4096    /* how many pages in the virtual addr space */
#endif

#define printf        printk

#define INIT_PID           1    /* init's process id number */
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/mm/type.h
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* If there were any type definitions local to the Memory Manager, they would
 * be here.  This file is included only for symmetry with the kernel and File
 * System, which do have some local type definitions.
 */

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Function prototypes. */

struct mproc;           /* need types outside of parameter list --kub */
struct stat;

/* alloc.c */
_PROTOTYPE( phys_clicks alloc_mem, (phys_clicks clicks)                 );
_PROTOTYPE( void free_mem, (phys_clicks base, phys_clicks clicks)       );
_PROTOTYPE( phys_clicks max_hole, (void)                                );
_PROTOTYPE( void mem_init, (phys_clicks *total, phys_clicks *free)      );
_PROTOTYPE( phys_clicks mem_left, (void)                                );
_PROTOTYPE( int do_brk3, (void)                                         );

/* break.c */
_PROTOTYPE( int adjust, (struct mproc *rmp,
                        vir_clicks data_clicks, vir_bytes sp)           );
_PROTOTYPE( int do_brk, (void)                                          );
_PROTOTYPE( int size_ok, (int file_type, vir_clicks tc, vir_clicks dc,
                        vir_clicks sc, vir_clicks dvir, vir_clicks s_vir) );

/* exec.c */
_PROTOTYPE( int do_exec, (void)                                         );
_PROTOTYPE( struct mproc *find_share, (struct mproc *mp_ign, Ino_t ino,
                        Dev_t dev, time_t ctime)                        );

/* forkexit.c */
_PROTOTYPE( int do_fork, (void)                                         );
_PROTOTYPE( int do_mm_exit, (void)                                      );
_PROTOTYPE( int do_waitpid, (void)                                      );
_PROTOTYPE( void mm_exit, (struct mproc *rmp, int exit_status)          );

/* getset.c */
_PROTOTYPE( int do_getset, (void)                                       );

/* main.c */
_PROTOTYPE( void main, (void)                                           );

#if (MACHINE == MACINTOSH)
_PROTOTYPE( phys_clicks start_click, (void)                             );
#endif

_PROTOTYPE( void reply, (int proc_nr, int result, int res2, char *respt));

/* putk.c */
_PROTOTYPE( void putk, (int c)                                          );

/* signal.c */
_PROTOTYPE( int do_alarm, (void)                                        );
_PROTOTYPE( int do_kill, (void)                                         );
_PROTOTYPE( int do_ksig, (void)                                         );
_PROTOTYPE( int do_pause, (void)                                        );
_PROTOTYPE( int set_alarm, (int proc_nr, int sec)                       );
_PROTOTYPE( int check_sig, (pid_t proc_id, int signo)                   );
_PROTOTYPE( void sig_proc, (struct mproc *rmp, int sig_nr)              );
_PROTOTYPE( int do_sigaction, (void)                                    );
```

```
_PROTOTYPE( int do_sigpending, (void)                                   );
_PROTOTYPE( int do_sigprocmask, (void)                                  );
_PROTOTYPE( int do_sigreturn, (void)                                    );
_PROTOTYPE( int do_sigsuspend, (void)                                   );
_PROTOTYPE( int do_reboot, (void)                                       );

/* trace.c */
_PROTOTYPE( int do_trace, (void)                                        );
_PROTOTYPE( void stop_proc, (struct mproc *rmp, int sig_nr)             );

/* utility.c */
_PROTOTYPE( int allowed, (char *name_buf, struct stat *s_buf, int mask) );
_PROTOTYPE( int no_sys, (void)                                          );
_PROTOTYPE( void panic, (char *format, int num)                         );
_PROTOTYPE( void tell_fs, (int what, int p1, int p2, int p3)            );
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* EXTERN should be extern except in table.c */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

/* Global variables. */
EXTERN struct mproc *mp;        /* ptr to 'mproc' slot of current process */
EXTERN int dont_reply;          /* normally 0; set to 1 to inhibit reply */
EXTERN int procs_in_use;        /* how many processes are marked as IN_USE */

/* The parameters of the call are kept here. */
EXTERN message mm_in;           /* the incoming message itself is kept here. */
EXTERN message mm_out;          /* the reply message is built up here. */
EXTERN int who;                 /* caller's proc number */
EXTERN int mm_call;             /* system call number */

/* The following variables are used for returning results to the caller. */
EXTERN int err_code;            /* temporary storage for error number */
EXTERN int result2;             /* secondary result */
EXTERN char *res_ptr;           /* result, if pointer */

extern _PROTOTYPE (int (*call_vec[]), (void) ); /* system call handlers */
extern char core_name[];        /* file name where core images are produced */
EXTERN sigset_t core_sset;      /* which signals cause core images */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/mproc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This table has one slot per process.  It contains all the memory management
 * information for each process.  Among other things, it defines the text, data
 * and stack segments, uids and gids, and various flags.  The kernel and file
 * systems have tables that are also indexed by process, with the contents
 * of corresponding slots referring to the same process in all three.
 */

EXTERN struct mproc {
  struct mem_map mp_seg[NR_SEGS];/* points to text, data, stack */
  char mp_exitstatus;           /* storage for status when process exits */
  char mp_sigstatus;            /* storage for signal # for killed procs */
  pid_t mp_pid;                 /* process id */
  pid_t mp_procgrp;             /* pid of process group (used for signals) */
  pid_t mp_wpid;                /* pid this process is waiting for */
  int mp_parent;                /* index of parent process */

  /* Real and effective uids and gids. */
  uid_t mp_realuid;             /* process' real uid */
  uid_t mp_effuid;              /* process' effective uid */
  gid_t mp_realgid;             /* process' real gid */
  gid_t mp_effgid;              /* process' effective gid */

  /* File identification for sharing. */
  ino_t mp_ino;                 /* inode number of file */
  dev_t mp_dev;                 /* device number of file system */
  time_t mp_ctime;              /* inode changed time */

  /* Signal handling information. */
  sigset_t mp_ignore;           /* 1 means ignore the signal, 0 means don't */
  sigset_t mp_catch;            /* 1 means catch the signal, 0 means don't */
  sigset_t mp_sigmask;          /* signals to be blocked */
  sigset_t mp_sigmask2;         /* saved copy of mp_sigmask */
  sigset_t mp_sigpending;       /* signals being blocked */
  struct sigaction mp_sigact[_NSIG + 1]; /* as in sigaction(2) */
  vir_bytes mp_sigreturn;       /* address of C library __sigreturn function */

  /* Backwards compatibility for signals. */
  sighandler_t mp_func;         /* all sigs vectored to a single user fcn */

  unsigned mp_flags;            /* flag bits */
  vir_bytes mp_procargs;        /* ptr to proc's initial stack arguments */
} mproc[NR_PROCS];

/* Flag values */
#define IN_USE            001   /* set when 'mproc' slot in use */
#define WAITING           002   /* set by WAIT system call */
#define HANGING           004   /* set by EXIT system call */
#define PAUSED            010   /* set by PAUSE system call */
#define ALARM_ON          020   /* set when SIGALRM timer started */
#define SEPARATE          040   /* set if file is separate I & D space */
#define TRACED           0100   /* set if process is to be traced */
#define STOPPED          0200   /* set if process stopped for tracing */
#define SIGSUSPENDED     0400   /* set by SIGSUSPEND system call */

#define NIL_MPROC ((struct mproc *) 0)
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/param.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The following names are synonyms for the variables in the input message. */
#define addr            mm_in.m1_p1
#define exec_name       mm_in.m1_p1
#define exec_len        mm_in.m1_i1
#define func            mm_in.m6_f1
#define grpid           (gid_t) mm_in.m1_i1
#define namelen         mm_in.m1_i1
#define pid             mm_in.m1_i1
#define seconds         mm_in.m1_i1
#define sig             mm_in.m6_i1
#define stack_bytes     mm_in.m1_i2
#define stack_ptr       mm_in.m1_p2
#define status          mm_in.m1_i1
#define usr_id          (uid_t) mm_in.m1_i1
#define request         mm_in.m2_i2
#define taddr           mm_in.m2_l1
#define data            mm_in.m2_l2
#define sig_nr          mm_in.m1_i2
#define sig_nsa         mm_in.m1_p1
#define sig_osa         mm_in.m1_p2
#define sig_ret         mm_in.m1_p3
#define sig_set         mm_in.m2_l1
#define sig_how         mm_in.m2_i1
#define sig_flags       mm_in.m2_i2
#define sig_context     mm_in.m2_p1
#ifdef _SIGMESSAGE
#define sig_msg         mm_in.m1_i1
#endif
#define reboot_flag     mm_in.m1_i1
#define reboot_code     mm_in.m1_p1
#define reboot_size     mm_in.m1_i2

/* The following names are synonyms for the variables in the output message. */
#define reply_type      mm_out.m_type
#define reply_i1        mm_out.m2_i1
#define reply_p1        mm_out.m2_p1
#define ret_mask        mm_out.m2_l1

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/table.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#define _TABLE

#include "mm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
```

```
#include "param.h"

/* Miscellaneous */
char core_name[] = "core";      /* file name where core images are produced */

_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
        no_sys,         /*  0 = unused  */
        do_mm_exit,     /*  1 = exit    */
        do_fork,        /*  2 = fork    */
        no_sys,         /*  3 = read    */
        no_sys,         /*  4 = write   */
        no_sys,         /*  5 = open    */
        no_sys,         /*  6 = close   */
        do_waitpid,     /*  7 = wait    */
        no_sys,         /*  8 = creat   */
        no_sys,         /*  9 = link    */
        no_sys,         /* 10 = unlink  */
        do_waitpid,     /* 11 = waitpid */
        no_sys,         /* 12 = chdir   */
        no_sys,         /* 13 = time    */
        no_sys,         /* 14 = mknod   */
        no_sys,         /* 15 = chmod   */
        no_sys,         /* 16 = chown   */
        do_brk,         /* 17 = break   */
        no_sys,         /* 18 = stat    */
        no_sys,         /* 19 = lseek   */
        do_getset,      /* 20 = getpid  */
        no_sys,         /* 21 = mount   */
        no_sys,         /* 22 = umount  */
        do_getset,      /* 23 = setuid  */
        do_getset,      /* 24 = getuid  */
        no_sys,         /* 25 = stime   */
        do_trace,       /* 26 = ptrace  */
        do_alarm,       /* 27 = alarm   */
        no_sys,         /* 28 = fstat   */
        do_pause,       /* 29 = pause   */
        no_sys,         /* 30 = utime   */
        no_sys,         /* 31 = (stty)  */
        no_sys,         /* 32 = (gtty)  */
        no_sys,         /* 33 = access  */
        no_sys,         /* 34 = (nice)  */
        no_sys,         /* 35 = (ftime) */
        no_sys,         /* 36 = sync    */
        do_kill,        /* 37 = kill    */
        no_sys,         /* 38 = rename  */
        no_sys,         /* 39 = mkdir   */
        no_sys,         /* 40 = rmdir   */
        no_sys,         /* 41 = dup     */
        no_sys,         /* 42 = pipe    */
        no_sys,         /* 43 = times   */
        no_sys,         /* 44 = (prof)  */
        no_sys,         /* 45 = unused  */
        do_getset,      /* 46 = setgid  */
        do_getset,      /* 47 = getgid  */
        no_sys,         /* 48 = (signal)*/
        no_sys,         /* 49 = unused  */
        no_sys,         /* 50 = unused  */
        no_sys,         /* 51 = (acct)  */
        no_sys,         /* 52 = (phys)  */
        no_sys,         /* 53 = (lock)  */
```

```
        no_sys,         /* 54 = ioctl   */
        no_sys,         /* 55 = fcntl   */
        no_sys,         /* 56 = (mpx)   */
        no_sys,         /* 57 = unused  */
        no_sys,         /* 58 = unused  */
        do_exec,        /* 59 = execve  */
        no_sys,         /* 60 = umask   */
        no_sys,         /* 61 = chroot  */
        do_getset,      /* 62 = setsid  */
        do_getset,      /* 63 = getpgrp */

        do_ksig,        /* 64 = KSIG: signals originating in the kernel */
        no_sys,         /* 65 = UNPAUSE */
        no_sys,         /* 66 = unused  */
        no_sys,         /* 67 = REVIVE  */
        no_sys,         /* 68 = TASK_REPLY  */
        no_sys,         /* 69 = unused  */
        no_sys,         /* 70 = unused  */
        do_sigaction,   /* 71 = sigaction   */
        do_sigsuspend,  /* 72 = sigsuspend  */
        do_sigpending,  /* 73 = sigpending  */
        do_sigprocmask, /* 74 = sigprocmask */
        do_sigreturn,   /* 75 = sigreturn   */
        do_reboot,      /* 76 = reboot  */
};
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/mm/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the main program of the memory manager and some related
 * procedures.  When MINIX starts up, the kernel runs for a little while,
 * initializing itself and its tasks, and then it runs MM and FS.  Both MM
 * and FS initialize themselves as far as they can.  FS then makes a call to
 * MM, because MM has to wait for FS to acquire a RAM disk.  MM asks the
 * kernel for all free memory and starts serving requests.
 *
 * The entry points into this file are:
 *   main:      starts MM running
 *   reply:     reply to a process making an MM system call
 */

#include "mm.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include "mproc.h"
#include "param.h"

FORWARD _PROTOTYPE( void get_work, (void)                               );
FORWARD _PROTOTYPE( void mm_init, (void)                                );

/*===========================================================================*
```

```
 *                              main                                        *
 *===========================================================================*/
PUBLIC void main()
{
/* Main routine of the memory manager. */

  int error;

  mm_init();                    /* initialize memory manager tables */

  /* This is MM's main loop-  get work and do it, forever and forever. */
  while (TRUE) {
        /* Wait for message. */
        get_work();             /* wait for an MM system call */
        mp = &mproc[who];

        /* Set some flags. */
        error = OK;
        dont_reply = FALSE;
        err_code = -999;

        /* If the call number is valid, perform the call. */
        if (mm_call < 0 || mm_call >= NCALLS)
                error = EBADCALL;
        else
                error = (*call_vec[mm_call])();

        /* Send the results back to the user to indicate completion. */
        if (dont_reply) continue;       /* no reply for EXIT and WAIT */
        if (mm_call == EXEC && error == OK) continue;
        reply(who, error, result2, res_ptr);
  }
}


/*===========================================================================*
 *                              get_work                                    *
 *===========================================================================*/
PRIVATE void get_work()
{
/* Wait for the next message and extract useful information from it. */

  if (receive(ANY, &mm_in) != OK) panic("MM receive error", NO_NUM);
  who = mm_in.m_source;         /* who sent the message */
  mm_call = mm_in.m_type;       /* system call number */
}


/*===========================================================================*
 *                              reply                                       *
 *===========================================================================*/
PUBLIC void reply(proc_nr, result, res2, respt)
int proc_nr;                    /* process to reply to */
int result;                     /* result of the call (usually OK or error #)*/
int res2;                       /* secondary result */
char *respt;                    /* result if pointer */
{
/* Send a reply to a user process. */

  register struct mproc *proc_ptr;
```

```

  proc_ptr = &mproc[proc_nr];
  /*
   * To make MM robust, check to see if destination is still alive.  This
   * validy check must be skipped if the caller is a task.
   */
  if ((who >=0) && ((proc_ptr->mp_flags&IN_USE) == 0 ||
        (proc_ptr->mp_flags&HANGING))) return;

  reply_type = result;
  reply_i1 = res2;
  reply_p1 = respt;
  if (send(proc_nr, &mm_out) != OK) panic("MM can't reply", NO_NUM);
}


/*===========================================================================*
 *                              mm_init                                      *
 *===========================================================================*/
PRIVATE void mm_init()
{
/* Initialize the memory manager. */

  static char core_sigs[] = {
        SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
        SIGEMT, SIGFPE, SIGUSR1, SIGSEGV,
        SIGUSR2, 0 };
  register int proc_nr;
  register struct mproc *rmp;
  register char *sig_ptr;
  phys_clicks ram_clicks, total_clicks, minix_clicks, free_clicks, dummy;
  message mess;
  struct mem_map kernel_map[NR_SEGS];
  int mem;

  /* Build the set of signals which cause core dumps. Do it the Posix
   * way, so no knowledge of bit positions is needed.
   */
  sigemptyset(&core_sset);
  for (sig_ptr = core_sigs; *sig_ptr != 0; sig_ptr++)
        sigaddset(&core_sset, *sig_ptr);

  /* Get the memory map of the kernel to see how much memory it uses,
   * including the gap between address 0 and the start of the kernel.
   */
  sys_getmap(SYSTASK, kernel_map);
  minix_clicks = kernel_map[S].mem_phys + kernel_map[S].mem_len;

  /* Initialize MM's tables. */
  for (proc_nr = 0; proc_nr <= INIT_PROC_NR; proc_nr++) {
        rmp = &mproc[proc_nr];
        rmp->mp_flags |= IN_USE;
        sys_getmap(proc_nr, rmp->mp_seg);
        if (rmp->mp_seg[T].mem_len != 0) rmp->mp_flags |= SEPARATE;
        minix_clicks += (rmp->mp_seg[S].mem_phys + rmp->mp_seg[S].mem_len)
                                - rmp->mp_seg[T].mem_phys;
  }
  mproc[INIT_PROC_NR].mp_pid = INIT_PID;
  sigemptyset(&mproc[INIT_PROC_NR].mp_ignore);
  sigemptyset(&mproc[INIT_PROC_NR].mp_catch);
```

```
  procs_in_use = LOW_USER + 1;

  /* Wait for FS to send a message telling the RAM disk size then go "on-line".
   */
  if (receive(FS_PROC_NR, &mess) != OK)
        panic("MM can't obtain RAM disk size from FS", NO_NUM);

  ram_clicks = mess.m1_i1;

  /* Initialize tables to all physical mem. */
  mem_init(&total_clicks, &free_clicks);

  /* Print memory information. */
  printf("\nMemory size =%5dK   ", click_to_round_k(total_clicks));
  printf("MINIX =%4dK   ", click_to_round_k(minix_clicks));
  printf("RAM disk =%5dK   ", click_to_round_k(ram_clicks));
  printf("Available =%5dK\n\n", click_to_round_k(free_clicks));

  /* Tell FS to continue. */
  if (send(FS_PROC_NR, &mess) != OK)
        panic("MM can't sync up with FS", NO_NUM);

  /* Tell the memory task where my process table is for the sake of ps(1). */
  if ((mem = open("/dev/mem", O_RDWR)) != -1) {
        ioctl(mem, MIOCSPSINFO, (void *) mproc);
        close(mem);
  }
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/forkexit.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file deals with creating processes (via FORK) and deleting them (via
 * EXIT/WAIT).  When a process forks, a new slot in the 'mproc' table is
 * allocated for it, and a copy of the parent's core image is made for the
 * child.  Then the kernel and file system are informed.  A process is removed
 * from the 'mproc' table when two events have occurred: (1) it has exited or
 * been killed by a signal, and (2) the parent has done a WAIT.  If the process
 * exits first, it continues to occupy a slot until the parent does a WAIT.
 *
 * The entry points into this file are:
 *   do_fork:    perform the FORK system call
 *   do_mm_exit: perform the EXIT system call (by calling mm_exit())
 *   mm_exit:    actually do the exiting
 *   do_wait:    perform the WAITPID or WAIT system call
 */


#include "mm.h"
#include <sys/wait.h>
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define LAST_FEW            2   /* last few slots reserved for superuser */

```

```
PRIVATE pid_t next_pid = INIT_PID+1;    /* next pid to be assigned */

FORWARD _PROTOTYPE (void cleanup, (register struct mproc *child) );

/*===========================================================================*
 *                              do_fork                                      *
 *===========================================================================*/
PUBLIC int do_fork()
{
/* The process pointed to by 'mp' has forked.  Create a child process. */

  register struct mproc *rmp;   /* pointer to parent */
  register struct mproc *rmc;   /* pointer to child */
  int i, child_nr, t;
  phys_clicks prog_clicks, child_base = 0;
  phys_bytes prog_bytes, parent_abs, child_abs; /* Intel only */

 /* If tables might fill up during FORK, don't even start since recovery half
  * way through is such a nuisance.
  */
  rmp = mp;
  if (procs_in_use == NR_PROCS) return(EAGAIN);
  if (procs_in_use >= NR_PROCS-LAST_FEW && rmp->mp_effuid != 0)return(EAGAIN);

  /* Determine how much memory to allocate.  Only the data and stack need to
   * be copied, because the text segment is either shared or of zero length.
   */
  prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
  prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  prog_bytes = (phys_bytes) prog_clicks << CLICK_SHIFT;
  if ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) return(EAGAIN);

  /* Create a copy of the parent's core image for the child. */
  child_abs = (phys_bytes) child_base << CLICK_SHIFT;
  parent_abs = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
  i = sys_copy(ABS, 0, parent_abs, ABS, 0, child_abs, prog_bytes);
  if (i < 0) panic("do_fork can't copy", i);

  /* Find a slot in 'mproc' for the child process.  A slot must exist. */
  for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
        if ( (rmc->mp_flags & IN_USE) == 0) break;

  /* Set up the child and its memory map; copy its 'mproc' slot from parent. */
  child_nr = (int)(rmc - mproc);        /* slot number of the child */
  procs_in_use++;
  *rmc = *rmp;                  /* copy parent's process slot to child's */

  rmc->mp_parent = who;         /* record child's parent */
  rmc->mp_flags &= ~TRACED;     /* child does not inherit trace status */
  /* A separate I&D child keeps the parents text segment.  The data and stack
   * segments must refer to the new copy.
   */
  if (!(rmc->mp_flags & SEPARATE)) rmc->mp_seg[T].mem_phys = child_base;
  rmc->mp_seg[D].mem_phys = child_base;
  rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
                        (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  rmc->mp_exitstatus = 0;
  rmc->mp_sigstatus = 0;

  /* Find a free pid for the child and put it in the table. */
```

```
  do {
        t = 0;                  /* 't' = 0 means pid still free */
        next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PID + 1);
        for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
                if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
                        t = 1;
                        break;
                }
        rmc->mp_pid = next_pid; /* assign pid to child */
  } while (t);

  /* Tell kernel and file system about the (now successful) FORK. */
  sys_fork(who, child_nr, rmc->mp_pid, child_base); /* child_base is 68K only*/
  tell_fs(FORK, who, child_nr, rmc->mp_pid);

  /* Report child's memory map to kernel. */
  sys_newmap(child_nr, rmc->mp_seg);

  /* Reply to child to wake it up. */
  reply(child_nr, 0, 0, NIL_PTR);
  return(next_pid);              /* child's pid */
}


/*===========================================================================*
 *                              do_mm_exit                                   *
 *===========================================================================*/
PUBLIC int do_mm_exit()
{
/* Perform the exit(status) system call. The real work is done by mm_exit(),
 * which is also called when a process is killed by a signal.
 */

  mm_exit(mp, status);
  dont_reply = TRUE;            /* don't reply to newly terminated process */
  return(OK);                   /* pro forma return code */
}


/*===========================================================================*
 *                              mm_exit                                      *
 *===========================================================================*/
PUBLIC void mm_exit(rmp, exit_status)
register struct mproc *rmp;     /* pointer to the process to be terminated */
int exit_status;                /* the process' exit status (for parent) */
{
/* A process is done.  Release most of the process' possessions.  If its
 * parent is waiting, release the rest, else hang.
 */

  register int proc_nr;
  int parent_waiting, right_child;
  pid_t pidarg, procgrp;
  phys_clicks base, size, s;            /* base and size used on 68000 only */

  proc_nr = (int) (rmp - mproc);        /* get process slot number */

  /* Remember a session leader's process group. */
  procgrp = (rmp->mp_pid == mp->mp_procgrp) ? mp->mp_procgrp : 0;

```

```
  /* If the exited process has a timer pending, kill it. */
  if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);

  /* Tell the kernel and FS that the process is no longer runnable. */
  tell_fs(EXIT, proc_nr, 0, 0);  /* file system can free the proc slot */
  sys_xit(rmp->mp_parent, proc_nr, &base, &size);

  /* Release the memory occupied by the child. */
  if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
        /* No other process shares the text segment, so free it. */
        free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
  }
  /* Free the data and stack segments. */
  free_mem(rmp->mp_seg[D].mem_phys,
      rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);

  /* The process slot can only be freed if the parent has done a WAIT. */
  rmp->mp_exitstatus = (char) exit_status;
  pidarg = mproc[rmp->mp_parent].mp_wpid;       /* who's being waited for? */
  parent_waiting = mproc[rmp->mp_parent].mp_flags & WAITING;
  if (pidarg == -1 || pidarg == rmp->mp_pid || -pidarg == rmp->mp_procgrp)
        right_child = TRUE;             /* child meets one of the 3 tests */
  else
        right_child = FALSE;            /* child fails all 3 tests */
  if (parent_waiting && right_child)
        cleanup(rmp);                   /* tell parent and release child slot */
  else
        rmp->mp_flags |= HANGING;       /* parent not waiting, suspend child */

  /* If the process has children, disinherit them.  INIT is the new parent. */
  for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++) {
        if (rmp->mp_flags & IN_USE && rmp->mp_parent == proc_nr) {
                /* 'rmp' now points to a child to be disinherited. */
                rmp->mp_parent = INIT_PROC_NR;
                parent_waiting = mproc[INIT_PROC_NR].mp_flags & WAITING;
                if (parent_waiting && (rmp->mp_flags & HANGING)) cleanup(rmp);
        }
  }

  /* Send a hangup to the process' process group if it was a session leader. */
  if (procgrp != 0) check_sig(-procgrp, SIGHUP);
}


/*===========================================================================*
 *                              do_waitpid                                   *
 *===========================================================================*/
PUBLIC int do_waitpid()
{
/* A process wants to wait for a child to terminate. If one is already waiting,
 * go clean it up and let this WAIT call terminate.  Otherwise, really wait.
 * Both WAIT and WAITPID are handled by this code.
 */

  register struct mproc *rp;
  int pidarg, options, children, res2;

  /* A process calling WAIT never gets a reply in the usual way via the
   * reply() in the main loop (unless WNOHANG is set or no qualifying child
   * exists).  If a child has already exited, the routine cleanup() sends
```

```
   * the reply to awaken the caller.
   */

  /* Set internal variables, depending on whether this is WAIT or WAITPID. */
  pidarg  = (mm_call == WAIT ? -1 : pid);       /* first param of waitpid */
  options = (mm_call == WAIT ?  0 : sig_nr);    /* third param of waitpid */
  if (pidarg == 0) pidarg = -mp->mp_procgrp;    /* pidarg < 0 ==> proc grp */

  /* Is there a child waiting to be collected? At this point, pidarg != 0:
   *    pidarg  >  0 means pidarg is pid of a specific process to wait for
   *    pidarg == -1 means wait for any child
   *    pidarg  < -1 means wait for any child whose process group = -pidarg
   */
  children = 0;
  for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
        if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who) {
                /* The value of pidarg determines which children qualify. */
                if (pidarg  > 0 && pidarg != rp->mp_pid) continue;
                if (pidarg < -1 && -pidarg != rp->mp_procgrp) continue;

                children++;             /* this child is acceptable */
                if (rp->mp_flags & HANGING) {
                        /* This child meets the pid test and has exited. */
                        cleanup(rp);    /* this child has already exited */
                        dont_reply = TRUE;
                        return(OK);
                }
                if ((rp->mp_flags & STOPPED) && rp->mp_sigstatus) {
                        /* This child meets the pid test and is being traced.*/
                        res2 =  0177 | (rp->mp_sigstatus << 8);
                        reply(who, rp->mp_pid, res2, NIL_PTR);
                        dont_reply = TRUE;
                        rp->mp_sigstatus = 0;
                        return(OK);
                }
        }
  }

  /* No qualifying child has exited.  Wait for one, unless none exists. */
  if (children > 0) {
        /* At least 1 child meets the pid test exists, but has not exited. */
        if (options & WNOHANG) return(0);    /* parent does not want to wait */
        mp->mp_flags |= WAITING;             /* parent wants to wait */
        mp->mp_wpid = (pid_t) pidarg;        /* save pid for later */
        dont_reply = TRUE;                   /* do not reply now though */
        return(OK);                          /* yes - wait for one to exit */
  } else {
        /* No child even meets the pid test.  Return error immediately. */
        return(ECHILD);                      /* no - parent has no children */
  }
}


/*===========================================================================*
 *                              cleanup                                      *
 *===========================================================================*/
PRIVATE void cleanup(child)
register struct mproc *child;   /* tells which process is exiting */
{
/* Finish off the exit of a process.  The process has exited or been killed
```

```
 * by a signal, and its parent is waiting.
 */

  int exitstatus;

  /* Wake up the parent. */
  exitstatus = (child->mp_exitstatus << 8) | (child->mp_sigstatus & 0377);
  reply(child->mp_parent, child->mp_pid, exitstatus, NIL_PTR);
  mproc[child->mp_parent].mp_flags &= ~WAITING; /* parent no longer waiting */

  /* Release the process table entry. */
  child->mp_flags = 0;
  procs_in_use--;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/mm/exec.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file handles the EXEC system call.  It performs the work as follows:
 *    - see if the permissions allow the file to be executed
 *    - read the header and extract the sizes
 *    - fetch the initial args and environment from the user space
 *    - allocate the memory for the new process
 *    - copy the initial stack from MM to the process
 *    - read in the text and data segments and copy to the process
 *    - take care of setuid and setgid bits
 *    - fix up 'mproc' table
 *    - tell kernel about EXEC
 *    - save offset to initial argc (for ps)
 *
 * The entry points into this file are:
 *   do_exec:    perform the EXEC system call
 *   find_share: find a process whose text segment can be shared
 */

#include "mm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <a.out.h>
#include <signal.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

FORWARD _PROTOTYPE( void load_seg, (int fd, int seg, vir_bytes seg_bytes) );
FORWARD _PROTOTYPE( int new_mem, (struct mproc *sh_mp, vir_bytes text_bytes,
                vir_bytes data_bytes, vir_bytes bss_bytes,
                vir_bytes stk_bytes, phys_bytes tot_bytes)              );
FORWARD _PROTOTYPE( void patch_ptr, (char stack [ARG_MAX ], vir_bytes base) );
FORWARD _PROTOTYPE( int read_header, (int fd, int *ft, vir_bytes *text_bytes,
                vir_bytes *data_bytes, vir_bytes *bss_bytes,
                phys_bytes *tot_bytes, long *sym_bytes, vir_clicks sc,
                vir_bytes *pc)                                          );


/*===========================================================================*
 *                              do_exec                                      *
 *===========================================================================*/
```

```
PUBLIC int do_exec()
{
/* Perform the execve(name, argv, envp) call.  The user library builds a
 * complete stack image, including pointers, args, environ, etc.  The stack
 * is copied to a buffer inside MM, and then to the new core image.
 */

  register struct mproc *rmp;
  struct mproc *sh_mp;
  int m, r, fd, ft, sn;
  static char mbuf[ARG_MAX];    /* buffer for stack and zeroes */
  static char name_buf[PATH_MAX]; /* the name of the file to exec */
  char *new_sp, *basename;
  vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
  phys_bytes tot_bytes;         /* total space for program, including gap */
  long sym_bytes;
  vir_clicks sc;
  struct stat s_buf;
  vir_bytes pc;

  /* Do some validity checks. */
  rmp = mp;
  stk_bytes = (vir_bytes) stack_bytes;
  if (stk_bytes > ARG_MAX) return(ENOMEM);      /* stack too big */
  if (exec_len <= 0 || exec_len > PATH_MAX) return(EINVAL);

  /* Get the exec file name and see if the file is executable. */
  src = (vir_bytes) exec_name;
  dst = (vir_bytes) name_buf;
  r = sys_copy(who, D, (phys_bytes) src,
                MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes) exec_len);
  if (r != OK) return(r);       /* file name not in user data segment */
  tell_fs(CHDIR, who, FALSE, 0);        /* switch to the user's FS environ. */
  fd = allowed(name_buf, &s_buf, X_BIT);        /* is file executable? */
  if (fd < 0) return(fd);       /* file was not executable */

  /* Read the file header and extract the segment sizes. */
  sc = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  m = read_header(fd, &ft, &text_bytes, &data_bytes, &bss_bytes,
                                        &tot_bytes, &sym_bytes, sc, &pc);
  if (m < 0) {
        close(fd);              /* something wrong with header */
        return(ENOEXEC);
  }

  /* Fetch the stack from the user before destroying the old core image. */
  src = (vir_bytes) stack_ptr;
  dst = (vir_bytes) mbuf;
  r = sys_copy(who, D, (phys_bytes) src,
                        MM_PROC_NR, D, (phys_bytes) dst, (phys_bytes)stk_bytes);
  if (r != OK) {
        close(fd);              /* can't fetch stack (e.g. bad virtual addr) */
        return(EACCES);
  }

  /* Can the process' text be shared with that of one already running? */
  sh_mp = find_share(rmp, s_buf.st_ino, s_buf.st_dev, s_buf.st_ctime);

  /* Allocate new memory and release old memory.  Fix map and tell kernel. */
  r = new_mem(sh_mp, text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes);
```

```
	if (r != OK) {
		close(fd);		/* insufficient core or program too big */
		return(r);
	}

	/* Save file identification to allow it to be shared. */
	rmp->mp_ino = s_buf.st_ino;
	rmp->mp_dev = s_buf.st_dev;
	rmp->mp_ctime = s_buf.st_ctime;

	/* Patch up stack and copy it from MM to new core image. */
	vsp = (vir_bytes) rmp->mp_seg[S].mem_vir << CLICK_SHIFT;
	vsp += (vir_bytes) rmp->mp_seg[S].mem_len << CLICK_SHIFT;
	vsp -= stk_bytes;
	patch_ptr(mbuf, vsp);
	src = (vir_bytes) mbuf;
	r = sys_copy(MM_PROC_NR, D, (phys_bytes) src,
				who, D, (phys_bytes) vsp, (phys_bytes)stk_bytes);
	if (r != OK) panic("do_exec stack copy err", NO_NUM);

	/* Read in text and data segments. */
	if (sh_mp != NULL) {
		lseek(fd, (off_t) text_bytes, SEEK_CUR);  /* shared: skip text */
	} else {
		load_seg(fd, T, text_bytes);
	}
	load_seg(fd, D, data_bytes);


	close(fd);			/* don't need exec file any more */

	/* Take care of setuid/setgid bits. */
	if ((rmp->mp_flags & TRACED) == 0) { /* suppress if tracing */
		if (s_buf.st_mode & I_SET_UID_BIT) {
			rmp->mp_effuid = s_buf.st_uid;
			tell_fs(SETUID,who, (int)rmp->mp_realuid, (int)rmp->mp_effuid);
		}
		if (s_buf.st_mode & I_SET_GID_BIT) {
			rmp->mp_effgid = s_buf.st_gid;
			tell_fs(SETGID,who, (int)rmp->mp_realgid, (int)rmp->mp_effgid);
		}
	}

	/* Save offset to initial argc (for ps) */
	rmp->mp_procargs = vsp;

	/* Fix 'mproc' fields, tell kernel that exec is done,  reset caught sigs. */
	for (sn = 1; sn <= _NSIG; sn++) {
		if (sigismember(&rmp->mp_catch, sn)) {
			sigdelset(&rmp->mp_catch, sn);
			rmp->mp_sigact[sn].sa_handler = SIG_DFL;
			sigemptyset(&rmp->mp_sigact[sn].sa_mask);
		}
	}

	rmp->mp_flags &= ~SEPARATE;	/* turn off SEPARATE bit */
	rmp->mp_flags |= ft;		/* turn it on for separate I & D files */
	new_sp = (char *) vsp;

	tell_fs(EXEC, who, 0, 0);	/* allow FS to handle FD_CLOEXEC files */
```

```

  /* System will save command line for debugging, ps(1) output, etc. */
  basename = strrchr(name_buf, '/');
  if (basename == NULL) basename = name_buf; else basename++;
  sys_exec(who, new_sp, rmp->mp_flags & TRACED, basename, pc);
  return(OK);
}


/*===========================================================================*
 *                              read_header                                  *
 *===========================================================================*/
PRIVATE int read_header(fd, ft, text_bytes, data_bytes, bss_bytes,
                                                tot_bytes, sym_bytes, sc, pc)
int fd;                         /* file descriptor for reading exec file */
int *ft;                        /* place to return ft number */
vir_bytes *text_bytes;          /* place to return text size */
vir_bytes *data_bytes;          /* place to return initialized data size */
vir_bytes *bss_bytes;           /* place to return bss size */
phys_bytes *tot_bytes;          /* place to return total size */
long *sym_bytes;                /* place to return symbol table size */
vir_clicks sc;                  /* stack size in clicks */
vir_bytes *pc;                  /* program entry point (initial PC) */
{
/* Read the header and extract the text, data, bss and total sizes from it. */

  int m, ct;
  vir_clicks tc, dc, s_vir, dvir;
  phys_clicks totc;
  struct exec hdr;              /* a.out header is read in here */

  /* Read the header and check the magic number.  The standard MINIX header
   * is defined in <a.out.h>.  It consists of 8 chars followed by 6 longs.
   * Then come 4 more longs that are not used here.
   *    Byte 0: magic number 0x01
   *    Byte 1: magic number 0x03
   *    Byte 2: normal = 0x10 (not checked, 0 is OK), separate I/D = 0x20
   *    Byte 3: CPU type, Intel 16 bit = 0x04, Intel 32 bit = 0x10,
   *            Motorola = 0x0B, Sun SPARC = 0x17
   *    Byte 4: Header length = 0x20
   *    Bytes 5-7 are not used.
   *
   *    Now come the 6 longs
   *    Bytes  8-11: size of text segments in bytes
   *    Bytes 12-15: size of initialized data segment in bytes
   *    Bytes 16-19: size of bss in bytes
   *    Bytes 20-23: program entry point
   *    Bytes 24-27: total memory allocated to program (text, data + stack)
   *    Bytes 28-31: size of symbol table in bytes
   * The longs are represented in a machine dependent order,
   * little-endian on the 8088, big-endian on the 68000.
   * The header is followed directly by the text and data segments, and the
   * symbol table (if any). The sizes are given in the header. Only the
   * text and data segments are copied into memory by exec. The header is
   * used here only. The symbol table is for the benefit of a debugger and
   * is ignored here.
   */

  if (read(fd, (char *) &hdr, A_MINHDR) != A_MINHDR) return(ENOEXEC);

```

```
  /* Check magic number, cpu type, and flags. */
  if (BADMAG(hdr)) return(ENOEXEC);
#if (CHIP == INTEL && _WORD_SIZE == 2)
  if (hdr.a_cpu != A_I8086) return(ENOEXEC);
#endif
#if (CHIP == INTEL && _WORD_SIZE == 4)
  if (hdr.a_cpu != A_I80386) return(ENOEXEC);
#endif
  if ((hdr.a_flags & ~(A_NSYM | A_EXEC | A_SEP)) != 0) return(ENOEXEC);

  *ft = ( (hdr.a_flags & A_SEP) ? SEPARATE : 0);    /* separate I & D or not */

  /* Get text and data sizes. */
  *text_bytes = (vir_bytes) hdr.a_text; /* text size in bytes */
  *data_bytes = (vir_bytes) hdr.a_data; /* data size in bytes */
  *bss_bytes  = (vir_bytes) hdr.a_bss;  /* bss size in bytes */
  *tot_bytes  = hdr.a_total;            /* total bytes to allocate for prog */
  *sym_bytes  = hdr.a_syms;             /* symbol table size in bytes */
  if (*tot_bytes == 0) return(ENOEXEC);

  if (*ft != SEPARATE) {

        /* If I & D space is not separated, it is all considered data. Text=0*/
        *data_bytes += *text_bytes;
        *text_bytes = 0;

  }
  *pc = hdr.a_entry;    /* initial address to start execution */

  /* Check to see if segment sizes are feasible. */
  tc = ((unsigned long) *text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  dc = (*data_bytes + *bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  totc = (*tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  if (dc >= totc) return(ENOEXEC);      /* stack must be at least 1 click */
  dvir = (*ft == SEPARATE ? 0 : tc);
  s_vir = dvir + (totc - sc);
  m = size_ok(*ft, tc, dc, sc, dvir, s_vir);
  ct = hdr.a_hdrlen & BYTE;             /* header length */
  if (ct > A_MINHDR) lseek(fd, (off_t) ct, SEEK_SET); /* skip unused hdr */
  return(m);
}


/*===========================================================================*
 *                              new_mem                                      *
 *===========================================================================*/
PRIVATE int new_mem(sh_mp, text_bytes, data_bytes,bss_bytes,stk_bytes,tot_bytes)
struct mproc *sh_mp;            /* text can be shared with this process */
vir_bytes text_bytes;           /* text segment size in bytes */
vir_bytes data_bytes;           /* size of initialized data in bytes */
vir_bytes bss_bytes;            /* size of bss in bytes */
vir_bytes stk_bytes;            /* size of initial stack segment in bytes */
phys_bytes tot_bytes;           /* total memory to allocate, including gap */
{
/* Allocate new memory and release the old memory.  Change the map and report
 * the new map to the kernel.  Zero the new core image's bss, gap and stack.
 */

  register struct mproc *rmp;
  vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks, tot_clicks;
```

```
  phys_clicks new_base;

  static char zero[1024];		/* used to zero bss */
  phys_bytes bytes, base, count, bss_offset;

  /* No need to allocate text if it can be shared. */
  if (sh_mp != NULL) text_bytes = 0;

  /* Acquire the new memory.  Each of the 4 parts: text, (data+bss), gap,
   * and stack occupies an integral number of clicks, starting at click
   * boundary.  The data and bss parts are run together with no space.
   */

  text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  gap_clicks = tot_clicks - data_clicks - stack_clicks;
  if ( (int) gap_clicks < 0) return(ENOMEM);

  /* Check to see if there is a hole big enough.  If so, we can risk first
   * releasing the old core image before allocating the new one, since we
   * know it will succeed.  If there is not enough, return failure.
   */
  if (text_clicks + tot_clicks > max_hole()) return(EAGAIN);

  /* There is enough memory for the new core image.  Release the old one. */
  rmp = mp;

  if (find_share(rmp, rmp->mp_ino, rmp->mp_dev, rmp->mp_ctime) == NULL) {
	/* No other process shares the text segment, so free it. */
	free_mem(rmp->mp_seg[T].mem_phys, rmp->mp_seg[T].mem_len);
  }
  /* Free the data and stack segments. */
  free_mem(rmp->mp_seg[D].mem_phys,
      rmp->mp_seg[S].mem_vir + rmp->mp_seg[S].mem_len - rmp->mp_seg[D].mem_vir);

  /* We have now passed the point of no return.  The old core image has been
   * forever lost.  The call must go through now.  Set up and report new map.
   */
  new_base = alloc_mem(text_clicks + tot_clicks);	/* new core image */
  if (new_base == NO_MEM) panic("MM hole list is inconsistent", NO_NUM);

  if (sh_mp != NULL) {
	/* Share the text segment. */
	rmp->mp_seg[T] = sh_mp->mp_seg[T];
  } else {
	rmp->mp_seg[T].mem_phys = new_base;
	rmp->mp_seg[T].mem_vir = 0;
	rmp->mp_seg[T].mem_len = text_clicks;
  }
  rmp->mp_seg[D].mem_phys = new_base + text_clicks;
  rmp->mp_seg[D].mem_vir = 0;
  rmp->mp_seg[D].mem_len = data_clicks;
  rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
  rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
  rmp->mp_seg[S].mem_len = stack_clicks;


  sys_newmap(who, rmp->mp_seg);   /* report new map to the kernel */
```

```

  /* Zero the bss, gap, and stack segment. */
  bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
  base = (phys_bytes) rmp->mp_seg[D].mem_phys << CLICK_SHIFT;
  bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
  base += bss_offset;
  bytes -= bss_offset;

  while (bytes > 0) {
        count = MIN(bytes, (phys_bytes) sizeof(zero));
        if (sys_copy(MM_PROC_NR, D, (phys_bytes) zero,
                                                ABS, 0, base, count) != OK) {
                panic("new_mem can't zero", NO_NUM);
        }
        base += count;
        bytes -= count;
  }

  return(OK);
}


/*===========================================================================*
 *                              patch_ptr                                    *
 *===========================================================================*/
PRIVATE void patch_ptr(stack, base)
char stack[ARG_MAX];    /* pointer to stack image within MM */
vir_bytes base;                 /* virtual address of stack base inside user */
{
/* When doing an exec(name, argv, envp) call, the user builds up a stack
 * image with arg and env pointers relative to the start of the stack.  Now
 * these pointers must be relocated, since the stack is not positioned at
 * address 0 in the user's address space.
 */

  char **ap, flag;
  vir_bytes v;

  flag = 0;                     /* counts number of 0-pointers seen */
  ap = (char **) stack;         /* points initially to 'nargs' */
  ap++;                         /* now points to argv[0] */
  while (flag < 2) {
        if (ap >= (char **) &stack[ARG_MAX]) return;    /* too bad */
        if (*ap != NIL_PTR) {
                v = (vir_bytes) *ap;    /* v is relative pointer */
                v += base;              /* relocate it */
                *ap = (char *) v;       /* put it back */
        } else {
                flag++;
        }
        ap++;
  }
}


/*===========================================================================*
 *                              load_seg                                     *
 *===========================================================================*/
PRIVATE void load_seg(fd, seg, seg_bytes)
int fd;                         /* file descriptor to read from */
```

```
int seg;                        /* T or D */
vir_bytes seg_bytes;            /* how big is the segment */
{
/* Read in text or data from the exec file and copy to the new core image.
 * This procedure is a little bit tricky.  The logical way to load a segment
 * would be to read it block by block and copy each block to the user space
 * one at a time.  This is too slow, so we do something dirty here, namely
 * send the user space and virtual address to the file system in the upper
 * 10 bits of the file descriptor, and pass it the user virtual address
 * instead of a MM address.  The file system extracts these parameters when
 * gets a read call from the memory manager, which is the only process that
 * is permitted to use this trick.  The file system then copies the whole
 * segment directly to user space, bypassing MM completely.
 */

  int new_fd, bytes;
  char *ubuf_ptr;

  new_fd = (who << 8) | (seg << 6) | fd;
  ubuf_ptr = (char *) ((vir_bytes)mp->mp_seg[seg].mem_vir << CLICK_SHIFT);
  while (seg_bytes != 0) {
        bytes = (INT_MAX / BLOCK_SIZE) * BLOCK_SIZE;
        if (seg_bytes < bytes)
                bytes = (int)seg_bytes;
        if (read(new_fd, ubuf_ptr, bytes) != bytes)
                break;          /* error */
        ubuf_ptr += bytes;
        seg_bytes -= bytes;
  }
}


/*===========================================================================*
 *                              find_share                                   *
 *===========================================================================*/
PUBLIC struct mproc *find_share(mp_ign, ino, dev, ctime)
struct mproc *mp_ign;           /* process that should not be looked at */
ino_t ino;                      /* parameters that uniquely identify a file */
dev_t dev;
time_t ctime;
{
/* Look for a process that is the file <ino, dev, ctime> in execution.  Don't
 * accidentally "find" mp_ign, because it is the process on whose behalf this
 * call is made.
 */
  struct mproc *sh_mp;

  for (sh_mp = &mproc[INIT_PROC_NR]; sh_mp < &mproc[NR_PROCS]; sh_mp++) {
        if ((sh_mp->mp_flags & (IN_USE | HANGING | SEPARATE))
                                        != (IN_USE | SEPARATE)) continue;
        if (sh_mp == mp_ign) continue;
        if (sh_mp->mp_ino != ino) continue;
        if (sh_mp->mp_dev != dev) continue;
        if (sh_mp->mp_ctime != ctime) continue;
        return sh_mp;
  }
  return(NULL);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  src/mm/break.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The MINIX model of memory allocation reserves a fixed amount of memory for
 * the combined text, data, and stack segments.  The amount used for a child
 * process created by FORK is the same as the parent had.  If the child does
 * an EXEC later, the new size is taken from the header of the file EXEC'ed.
 *
 * The layout in memory consists of the text segment, followed by the data
 * segment, followed by a gap (unused memory), followed by the stack segment.
 * The data segment grows upward and the stack grows downward, so each can
 * take memory from the gap.  If they meet, the process must be killed.  The
 * procedures in this file deal with the growth of the data and stack segments.
 *
 * The entry points into this file are:
 *   do_brk:      BRK/SBRK system calls to grow or shrink the data segment
 *   adjust:      see if a proposed segment adjustment is allowed
 *   size_ok:     see if the segment sizes are feasible
 */

#include "mm.h"
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define DATA_CHANGED       1    /* flag value when data segment size changed */
#define STACK_CHANGED      2    /* flag value when stack size changed */

/*===========================================================================*
 *                              do_brk                                       *
 *===========================================================================*/
PUBLIC int do_brk()
{
/* Perform the brk(addr) system call.
 *
 * The call is complicated by the fact that on some machines (e.g., 8088),
 * the stack pointer can grow beyond the base of the stack segment without
 * anybody noticing it.
 * The parameter, 'addr' is the new virtual address in D space.
 */

  register struct mproc *rmp;
  int r;
  vir_bytes v, new_sp;
  vir_clicks new_clicks;

  rmp = mp;
  v = (vir_bytes) addr;
  new_clicks = (vir_clicks) ( ((long) v + CLICK_SIZE - 1) >> CLICK_SHIFT);
  if (new_clicks < rmp->mp_seg[D].mem_vir) {
        res_ptr = (char *) -1;
        return(ENOMEM);
  }
  new_clicks -= rmp->mp_seg[D].mem_vir;
  sys_getsp(who, &new_sp);      /* ask kernel for current sp value */
  r = adjust(rmp, new_clicks, new_sp);
  res_ptr = (r == OK ? addr : (char *) -1);
  return(r);                    /* return new address or -1 */
```

```
}


/*===========================================================================*
 *                              adjust                                       *
 *===========================================================================*/
PUBLIC int adjust(rmp, data_clicks, sp)
register struct mproc *rmp;     /* whose memory is being adjusted? */
vir_clicks data_clicks;         /* how big is data segment to become? */
vir_bytes sp;                   /* new value of sp */
{
/* See if data and stack segments can coexist, adjusting them if need be.
 * Memory is never allocated or freed.  Instead it is added or removed from the
 * gap between data segment and stack segment.  If the gap size becomes
 * negative, the adjustment of data or stack fails and ENOMEM is returned.
 */

  register struct mem_map *mem_sp, *mem_dp;
  vir_clicks sp_click, gap_base, lower, old_clicks;
  int changed, r, ft;
  long base_of_stack, delta;    /* longs avoid certain problems */

  mem_dp = &rmp->mp_seg[D];     /* pointer to data segment map */
  mem_sp = &rmp->mp_seg[S];     /* pointer to stack segment map */
  changed = 0;                  /* set when either segment changed */

  if (mem_sp->mem_len == 0) return(OK); /* don't bother init */

  /* See if stack size has gone negative (i.e., sp too close to 0xFFFF...) */
  base_of_stack = (long) mem_sp->mem_vir + (long) mem_sp->mem_len;
  sp_click = sp >> CLICK_SHIFT; /* click containing sp */
  if (sp_click >= base_of_stack) return(ENOMEM);        /* sp too high */

  /* Compute size of gap between stack and data segments. */
  delta = (long) mem_sp->mem_vir - (long) sp_click;
  lower = (delta > 0 ? sp_click : mem_sp->mem_vir);

  /* Add a safety margin for future stack growth. Impossible to do right. */
#define SAFETY_BYTES  (384 * sizeof(char *))
#define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
  gap_base = mem_dp->mem_vir + data_clicks + SAFETY_CLICKS;
  if (lower < gap_base) return(ENOMEM); /* data and stack collided */

  /* Update data length (but not data orgin) on behalf of brk() system call. */
  old_clicks = mem_dp->mem_len;
  if (data_clicks != mem_dp->mem_len) {
        mem_dp->mem_len = data_clicks;
        changed |= DATA_CHANGED;
  }

  /* Update stack length and origin due to change in stack pointer. */
  if (delta > 0) {
        mem_sp->mem_vir -= delta;
        mem_sp->mem_phys -= delta;
        mem_sp->mem_len += delta;
        changed |= STACK_CHANGED;
  }

  /* Do the new data and stack segment sizes fit in the address space? */
  ft = (rmp->mp_flags & SEPARATE);
```

```
  r = size_ok(ft, rmp->mp_seg[T].mem_len, rmp->mp_seg[D].mem_len,
       rmp->mp_seg[S].mem_len, rmp->mp_seg[D].mem_vir, rmp->mp_seg[S].mem_vir);
  if (r == OK) {
        if (changed) sys_newmap((int)(rmp - mproc), rmp->mp_seg);
        return(OK);
  }

  /* New sizes don't fit or require too many page/segment registers. Restore.*/
  if (changed & DATA_CHANGED) mem_dp->mem_len = old_clicks;
  if (changed & STACK_CHANGED) {
        mem_sp->mem_vir += delta;
        mem_sp->mem_phys += delta;
        mem_sp->mem_len -= delta;
  }
  return(ENOMEM);
}


/*===========================================================================*
 *                              size_ok                                      *
 *===========================================================================*/
PUBLIC int size_ok(file_type, tc, dc, sc, dvir, s_vir)
int file_type;                  /* SEPARATE or 0 */
vir_clicks tc;                  /* text size in clicks */
vir_clicks dc;                  /* data size in clicks */
vir_clicks sc;                  /* stack size in clicks */
vir_clicks dvir;                /* virtual address for start of data seg */
vir_clicks s_vir;               /* virtual address for start of stack seg */
{
/* Check to see if the sizes are feasible and enough segmentation registers
 * exist.  On a machine with eight 8K pages, text, data, stack sizes of
 * (32K, 16K, 16K) will fit, but (33K, 17K, 13K) will not, even though the
 * former is bigger (64K) than the latter (63K).  Even on the 8088 this test
 * is needed, since the data and stack may not exceed 4096 clicks.
 */

#if (CHIP == INTEL && _WORD_SIZE == 2)
  int pt, pd, ps;               /* segment sizes in pages */

  pt = ( (tc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
  pd = ( (dc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;
  ps = ( (sc << CLICK_SHIFT) + PAGE_SIZE - 1)/PAGE_SIZE;

  if (file_type == SEPARATE) {
        if (pt > MAX_PAGES || pd + ps > MAX_PAGES) return(ENOMEM);
  } else {
        if (pt + pd + ps > MAX_PAGES) return(ENOMEM);
  }
#endif

  if (dvir + dc > s_vir) return(ENOMEM);

  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/signal.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file handles signals, which are asynchronous events and are generally
 * a messy and unpleasant business.  Signals can be generated by the KILL
 * system call, or from the keyboard (SIGINT) or from the clock (SIGALRM).
 * In all cases control eventually passes to check_sig() to see which processes
 * can be signaled.  The actual signaling is done by sig_proc().
 *
 * The entry points into this file are:
 *   do_sigaction:   perform the SIGACTION system call
 *   do_sigpending:  perform the SIGPENDING system call
 *   do_sigprocmask: perform the SIGPROCMASK system call
 *   do_sigreturn:   perform the SIGRETURN system call
 *   do_sigsuspend:  perform the SIGSUSPEND system call
 *   do_kill:   perform the KILL system call
 *   do_ksig:   accept a signal originating in the kernel (e.g., SIGINT)
 *   do_alarm:  perform the ALARM system call by calling set_alarm()
 *   set_alarm: tell the clock task to start or stop a timer
 *   do_pause:  perform the PAUSE system call
 *   do_reboot: kill all processes, then reboot system
 *   sig_proc:  interrupt or terminate a signaled process
 *   check_sig: check which processes to signal with sig_proc()
 */

#include "mm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <signal.h>
#include <sys/sigcontext.h>
#include <string.h>
#include "mproc.h"
#include "param.h"

#define CORE_MODE       0777    /* mode to use on core image files */
#define DUMPED          0200    /* bit set in status when core dumped */
#define DUMP_SIZE       ((INT_MAX / BLOCK_SIZE) * BLOCK_SIZE)
                                /* buffer size for core dumps */

FORWARD _PROTOTYPE( void check_pending, (void)                          );
FORWARD _PROTOTYPE( void dump_core, (struct mproc *rmp)                 );
FORWARD _PROTOTYPE( void unpause, (int pro)                             );


/*===========================================================================*
 *                              do_sigaction                                 *
 *===========================================================================*/
PUBLIC int do_sigaction()
{
  int r;
  struct sigaction svec;
  struct sigaction *svp;

  if (sig_nr == SIGKILL) return(OK);
  if (sig_nr < 1 || sig_nr > _NSIG) return (EINVAL);
  svp = &mp->mp_sigact[sig_nr];
  if ((struct sigaction *) sig_osa != (struct sigaction *) NULL) {
```

```
          r = sys_copy(MM_PROC_NR,D, (phys_bytes) svp,
                  who, D, (phys_bytes) sig_osa, (phys_bytes) sizeof(svec));
          if (r != OK) return(r);
    }

    if ((struct sigaction *) sig_nsa == (struct sigaction *) NULL) return(OK);

    /* Read in the sigaction structure. */
    r = sys_copy(who, D, (phys_bytes) sig_nsa,
                  MM_PROC_NR, D, (phys_bytes) &svec, (phys_bytes) sizeof(svec));
    if (r != OK) return(r);

    if (svec.sa_handler == SIG_IGN) {
          sigaddset(&mp->mp_ignore, sig_nr);
          sigdelset(&mp->mp_sigpending, sig_nr);
          sigdelset(&mp->mp_catch, sig_nr);
    } else {
          sigdelset(&mp->mp_ignore, sig_nr);
          if (svec.sa_handler == SIG_DFL)
                  sigdelset(&mp->mp_catch, sig_nr);
          else
                  sigaddset(&mp->mp_catch, sig_nr);
    }
    mp->mp_sigact[sig_nr].sa_handler = svec.sa_handler;
    sigdelset(&svec.sa_mask, SIGKILL);
    mp->mp_sigact[sig_nr].sa_mask = svec.sa_mask;
    mp->mp_sigact[sig_nr].sa_flags = svec.sa_flags;
    mp->mp_sigreturn = (vir_bytes) sig_ret;
    return(OK);
  }

  /*===========================================================================*
   *                             do_sigpending                                 *
   *===========================================================================*/
  PUBLIC int do_sigpending()
  {
    ret_mask = (long) mp->mp_sigpending;
    return OK;
  }

  /*===========================================================================*
   *                             do_sigprocmask                                *
   *===========================================================================*/
  PUBLIC int do_sigprocmask()
  {
  /* Note that the library interface passes the actual mask in sigmask_set,
   * not a pointer to the mask, in order to save a sys_copy.  Similarly,
   * the old mask is placed in the return message which the library
   * interface copies (if requested) to the user specified address.
   *
   * The library interface must set SIG_INQUIRE if the 'act' argument
   * is NULL.
   */

    int i;

    ret_mask = (long) mp->mp_sigmask;

    switch (sig_how) {
        case SIG_BLOCK:
```

```
        sigdelset((sigset_t *)&sig_set, SIGKILL);
        for (i = 1; i < _NSIG; i++) {
                if (sigismember((sigset_t *)&sig_set, i))
                        sigaddset(&mp->mp_sigmask, i);
        }
        break;

      case SIG_UNBLOCK:
        for (i = 1; i < _NSIG; i++) {
                if (sigismember((sigset_t *)&sig_set, i))
                        sigdelset(&mp->mp_sigmask, i);
        }
        check_pending();
        break;

      case SIG_SETMASK:
        sigdelset((sigset_t *)&sig_set, SIGKILL);
        mp->mp_sigmask = (sigset_t)sig_set;
        check_pending();
        break;

      case SIG_INQUIRE:
        break;

      default:
        return(EINVAL);
        break;
  }
  return OK;
}

/*===========================================================================*
 *                              do_sigsuspend                                *
 *===========================================================================*/
PUBLIC int do_sigsuspend()
{
  mp->mp_sigmask2 = mp->mp_sigmask;     /* save the old mask */
  mp->mp_sigmask = (sigset_t) sig_set;
  sigdelset(&mp->mp_sigmask, SIGKILL);
  mp->mp_flags |= SIGSUSPENDED;
  dont_reply = TRUE;
  check_pending();
  return OK;
}


/*===========================================================================*
 *                              do_sigreturn                                 *
 *===========================================================================*/
PUBLIC int do_sigreturn()
{
/* A user signal handler is done.  Restore context and check for
 * pending unblocked signals.
 */

  int r;

  mp->mp_sigmask = (sigset_t) sig_set;
  sigdelset(&mp->mp_sigmask, SIGKILL);

```

```
  r = sys_sigreturn(who, (vir_bytes)sig_context, sig_flags);
  check_pending();
  return(r);
}

/*===========================================================================*
 *                              do_kill                                      *
 *===========================================================================*/
PUBLIC int do_kill()
{
/* Perform the kill(pid, signo) system call. */

  return check_sig(pid, sig_nr);
}


/*===========================================================================*
 *                              do_ksig                                      *
 *===========================================================================*/
PUBLIC int do_ksig()
{
/* Certain signals, such as segmentation violations and DEL, originate in the
 * kernel.  When the kernel detects such signals, it sets bits in a bit map.
 * As soon as MM is awaiting new work, the kernel sends MM a message containing
 * the process slot and bit map.  That message comes here.  The File System
 * also uses this mechanism to signal writing on broken pipes (SIGPIPE).
 */

  register struct mproc *rmp;
  int i, proc_nr;
  pid_t proc_id, id;
  sigset_t sig_map;

  /* Only kernel may make this call. */
  if (who != HARDWARE) return(EPERM);
  dont_reply = TRUE;            /* don't reply to the kernel */
  proc_nr = mm_in.SIG_PROC;
  rmp = &mproc[proc_nr];
  if ( (rmp->mp_flags & IN_USE) == 0 || (rmp->mp_flags & HANGING) ) return(OK);
  proc_id = rmp->mp_pid;
  sig_map = (sigset_t) mm_in.SIG_MAP;
  mp = &mproc[0];               /* pretend kernel signals are from MM */
  mp->mp_procgrp = rmp->mp_procgrp;     /* get process group right */

  /* Check each bit in turn to see if a signal is to be sent.  Unlike
   * kill(), the kernel may collect several unrelated signals for a
   * process and pass them to MM in one blow.  Thus loop on the bit
   * map. For SIGINT and SIGQUIT, use proc_id 0 to indicate a broadcast
   * to the recipient's process group.  For SIGKILL, use proc_id -1 to
   * indicate a systemwide broadcast.
   */
  for (i = 1; i <= _NSIG; i++) {
        if (!sigismember(&sig_map, i)) continue;
        switch (i) {
            case SIGINT:
            case SIGQUIT:
                id = 0; break;  /* broadcast to process group */
            case SIGKILL:
                id = -1; break; /* broadcast to all except INIT */
            case SIGALRM:
```

```
		/* Disregard SIGALRM when the target process has not
		 * requested an alarm.  This only applies for a KERNEL
		 * generated signal.
		 */
		if ((rmp->mp_flags & ALARM_ON) == 0) continue;
		rmp->mp_flags &= ~ALARM_ON;
		/* fall through */
	    default:
		id = proc_id;
		break;
	}
	check_sig(id, i);
	sys_endsig(proc_nr);	/* tell kernel it's done */
  }
  return(OK);
}


/*===========================================================================*
 *                              do_alarm                                     *
 *===========================================================================*/
PUBLIC int do_alarm()
{
/* Perform the alarm(seconds) system call. */

  return(set_alarm(who, seconds));
}


/*===========================================================================*
 *                              set_alarm                                    *
 *===========================================================================*/
PUBLIC int set_alarm(proc_nr, sec)
int proc_nr;			/* process that wants the alarm */
int sec;			/* how many seconds delay before the signal */
{
/* This routine is used by do_alarm() to set the alarm timer.  It is also used
 * to turn the timer off when a process exits with the timer still on.
 */

  message m_sig;
  int remaining;

  if (sec != 0)
	mproc[proc_nr].mp_flags |= ALARM_ON;
  else
	mproc[proc_nr].mp_flags &= ~ALARM_ON;

  /* Tell the clock task to provide a signal message when the time comes.
   *
   * Large delays cause a lot of problems.  First, the alarm system call
   * takes an unsigned seconds count and the library has cast it to an int.
   * That probably works, but on return the library will convert "negative"
   * unsigneds to errors.  Presumably no one checks for these errors, so
   * force this call through.  Second, If unsigned and long have the same
   * size, converting from seconds to ticks can easily overflow.  Finally,
   * the kernel has similar overflow bugs adding ticks.
   *
   * Fixing this requires a lot of ugly casts to fit the wrong interface
   * types and to avoid overflow traps.  DELTA_TICKS has the right type
```

```
   * (clock_t) although it is declared as long.  How can variables like
   * this be declared properly without combinatorial explosion of message
   * types?
   */
  m_sig.m_type = SET_ALARM;
  m_sig.CLOCK_PROC_NR = proc_nr;
  m_sig.DELTA_TICKS = (clock_t) (HZ * (unsigned long) (unsigned) sec);
  if ( (unsigned long) m_sig.DELTA_TICKS / HZ != (unsigned) sec)
        m_sig.DELTA_TICKS = LONG_MAX;   /* eternity (really CLOCK_T_MAX) */
  if (sendrec(CLOCK, &m_sig) != OK) panic("alarm er", NO_NUM);
  remaining = (int) m_sig.SECONDS_LEFT;
  if (remaining != m_sig.SECONDS_LEFT || remaining < 0)
        remaining = INT_MAX;    /* true value is not representable */
  return(remaining);
}


/*===========================================================================*
 *                              do_pause                                     *
 *===========================================================================*/
PUBLIC int do_pause()
{
/* Perform the pause() system call. */

  mp->mp_flags |= PAUSED;
  dont_reply = TRUE;
  return(OK);
}


/*=====================================================================*
 *                            do_reboot                                *
 *=====================================================================*/
PUBLIC int do_reboot()
{
  register struct mproc *rmp = mp;
  char monitor_code[64];

  if (rmp->mp_effuid != SUPER_USER)   return EPERM;

  switch (reboot_flag) {
  case RBT_HALT:
  case RBT_REBOOT:
  case RBT_PANIC:
  case RBT_RESET:
        break;
  case RBT_MONITOR:
        if (reboot_size > sizeof(monitor_code)) return EINVAL;
        memset(monitor_code, 0, sizeof(monitor_code));
        if (sys_copy(who, D, (phys_bytes) reboot_code,
                MM_PROC_NR, D, (phys_bytes) monitor_code,
                (phys_bytes) reboot_size) != OK) return EFAULT;
        if (monitor_code[sizeof(monitor_code)-1] != 0) return EINVAL;
        break;
  default:
        return EINVAL;
  }

  /* Kill all processes except init. */
  check_sig(-1, SIGKILL);
```

```

  tell_fs(EXIT, INIT_PROC_NR, 0, 0);    /* cleanup init */

  tell_fs(SYNC,0,0,0);

  sys_abort(reboot_flag, monitor_code);
  /* NOTREACHED */
}


/*===========================================================================*
 *                              sig_proc                                     *
 *===========================================================================*/
PUBLIC void sig_proc(rmp, signo)
register struct mproc *rmp;     /* pointer to the process to be signaled */
int signo;                      /* signal to send to process (1 to _NSIG) */
{
/* Send a signal to a process.  Check to see if the signal is to be caught,
 * ignored, or blocked.  If the signal is to be caught, coordinate with
 * KERNEL to push a sigcontext structure and a sigframe structure onto
 * the catcher's stack.  Also, KERNEL will reset the program counter and
 * stack pointer, so that when the process next runs, it will be executing
 * the signal handler.  When the signal handler returns,  sigreturn(2)
 * will be called.  Then KERNEL will restore the signal context from the
 * sigcontext structure.
 *
 * If there is insufficient stack space, kill the process.
 */

  vir_bytes new_sp;
  int slot;
  int sigflags;
  struct sigmsg sm;

  slot = (int) (rmp - mproc);
  if (!(rmp->mp_flags & IN_USE)) {
        printf("MM: signal %d sent to dead process %d\n", signo, slot);
        panic("", NO_NUM);
  }
  if (rmp->mp_flags & HANGING) {
        printf("MM: signal %d sent to HANGING process %d\n", signo, slot);
        panic("", NO_NUM);
  }
  if (rmp->mp_flags & TRACED && signo != SIGKILL) {
        /* A traced process has special handling. */
        unpause(slot);
        stop_proc(rmp, signo);  /* a signal causes it to stop */
        return;
  }
  /* Some signals are ignored by default. */
  if (sigismember(&rmp->mp_ignore, signo)) return;

  if (sigismember(&rmp->mp_sigmask, signo)) {
        /* Signal should be blocked. */
        sigaddset(&rmp->mp_sigpending, signo);
        return;
  }
  sigflags = rmp->mp_sigact[signo].sa_flags;
  if (sigismember(&rmp->mp_catch, signo)) {
        if (rmp->mp_flags & SIGSUSPENDED)
```

```
                sm.sm_mask = rmp->mp_sigmask2;
        else
                sm.sm_mask = rmp->mp_sigmask;
        sm.sm_signo = signo;
        sm.sm_sighandler = (vir_bytes) rmp->mp_sigact[signo].sa_handler;
        sm.sm_sigreturn = rmp->mp_sigreturn;
        sys_getsp(slot, &new_sp);
        sm.sm_stkptr = new_sp;

        /* Make room for the sigcontext and sigframe struct. */
        new_sp -= sizeof(struct sigcontext)
                                 + 3 * sizeof(char *) + 2 * sizeof(int);

        if (adjust(rmp, rmp->mp_seg[D].mem_len, new_sp) != OK)
                goto doterminate;

        rmp->mp_sigmask |= rmp->mp_sigact[signo].sa_mask;
        if (sigflags & SA_NODEFER)
                sigdelset(&rmp->mp_sigmask, signo);
        else
                sigaddset(&rmp->mp_sigmask, signo);

        if (sigflags & SA_RESETHAND) {
                sigdelset(&rmp->mp_catch, signo);
                rmp->mp_sigact[signo].sa_handler = SIG_DFL;
        }

        sys_sendsig(slot, &sm);
        sigdelset(&rmp->mp_sigpending, signo);
        /* If process is hanging on PAUSE, WAIT, SIGSUSPEND, tty, pipe, etc.,
         * release it.
         */
        unpause(slot);
        return;
  }
doterminate:
  /* Signal should not or cannot be caught.  Terminate the process. */
  rmp->mp_sigstatus = (char) signo;
  if (sigismember(&core_sset, signo)) {
        /* Switch to the user's FS environment and dump core. */
        tell_fs(CHDIR, slot, FALSE, 0);
        dump_core(rmp);
  }
  mm_exit(rmp, 0);              /* terminate process */
}


/*===========================================================================*
 *                              check_sig                                    *
 *===========================================================================*/
PUBLIC int check_sig(proc_id, signo)
pid_t proc_id;                  /* pid of proc to sig, or 0 or -1, or -pgrp */
int signo;                      /* signal to send to process (0 to _NSIG) */
{
/* Check to see if it is possible to send a signal.  The signal may have to be
 * sent to a group of processes.  This routine is invoked by the KILL system
 * call, and also when the kernel catches a DEL or other signal.
 */

  register struct mproc *rmp;
```

```
  int count;                    /* count # of signals sent */
  int error_code;

  if (signo < 0 || signo > _NSIG) return(EINVAL);

  /* Return EINVAL for attempts to send SIGKILL to INIT alone. */
  if (proc_id == INIT_PID && signo == SIGKILL) return(EINVAL);

  /* Search the proc table for processes to signal.  (See forkexit.c about
   * pid magic.)
   */
  count = 0;
  error_code = ESRCH;
  for (rmp = &mproc[INIT_PROC_NR]; rmp < &mproc[NR_PROCS]; rmp++) {
        if ( (rmp->mp_flags & IN_USE) == 0) continue;
        if (rmp->mp_flags & HANGING && signo != 0) continue;

        /* Check for selection. */
        if (proc_id > 0 && proc_id != rmp->mp_pid) continue;
        if (proc_id == 0 && mp->mp_procgrp != rmp->mp_procgrp) continue;
        if (proc_id == -1 && rmp->mp_pid == INIT_PID) continue;
        if (proc_id < -1 && rmp->mp_procgrp != -proc_id) continue;

        /* Check for permission. */
        if (mp->mp_effuid != SUPER_USER
            && mp->mp_realuid != rmp->mp_realuid
            && mp->mp_effuid != rmp->mp_realuid
            && mp->mp_realuid != rmp->mp_effuid
            && mp->mp_effuid != rmp->mp_effuid) {
                error_code = EPERM;
                continue;
        }

        count++;
        if (signo == 0) continue;

        /* 'sig_proc' will handle the disposition of the signal.  The
         * signal may be caught, blocked, ignored, or cause process
         * termination, possibly with core dump.
         */
        sig_proc(rmp, signo);

        if (proc_id > 0) break; /* only one process being signaled */
  }

  /* If the calling process has killed itself, don't reply. */
  if ((mp->mp_flags & IN_USE) == 0 || (mp->mp_flags & HANGING))
        dont_reply = TRUE;
  return(count > 0 ? OK : error_code);
}


/*===========================================================================*
 *                              check_pending                                *
 *===========================================================================*/
PRIVATE void check_pending()
{
  /* Check to see if any pending signals have been unblocked.  The
   * first such signal found is delivered.
   *
```

```
 * If multiple pending unmasked signals are found, they will be
 * delivered sequentially.
 *
 * There are several places in this file where the signal mask is
 * changed.  At each such place, check_pending() should be called to
 * check for newly unblocked signals.
 */

  int i;

  for (i = 1; i < _NSIG; i++) {
	if (sigismember(&mp->mp_sigpending, i) &&
		!sigismember(&mp->mp_sigmask, i)) {
		sigdelset(&mp->mp_sigpending, i);
		sig_proc(mp, i);
		break;
	}
  }
}


/*===========================================================================*
 *				unpause					     *
 *===========================================================================*/
PRIVATE void unpause(pro)
int pro;			/* which process number */
{
/* A signal is to be sent to a process.  If that process is hanging on a
 * system call, the system call must be terminated with EINTR.  Possible
 * calls are PAUSE, WAIT, READ and WRITE, the latter two for pipes and ttys.
 * First check if the process is hanging on an MM call.  If not, tell FS,
 * so it can check for READs and WRITEs from pipes, ttys and the like.
 */

  register struct mproc *rmp;

  rmp = &mproc[pro];

  /* Check to see if process is hanging on a PAUSE call. */
  if ( (rmp->mp_flags & PAUSED) && (rmp->mp_flags & HANGING) == 0) {
	rmp->mp_flags &= ~PAUSED;
	reply(pro, EINTR, 0, NIL_PTR);
	return;
  }

  /* Check to see if process is hanging on a WAIT call. */
  if ( (rmp->mp_flags & WAITING) && (rmp->mp_flags & HANGING) == 0) {
	rmp->mp_flags &= ~WAITING;
	reply(pro, EINTR, 0, NIL_PTR);
	return;
  }

  /* Check to see if process is hanging on a SIGSUSPEND call. */
  if ((rmp->mp_flags & SIGSUSPENDED) && (rmp->mp_flags & HANGING) == 0) {
	rmp->mp_flags &= ~SIGSUSPENDED;
	reply(pro, EINTR, 0, NIL_PTR);
	return;
  }

  /* Process is not hanging on an MM call.  Ask FS to take a look. */
```

```
        tell_fs(UNPAUSE, pro, 0, 0);
}


/*===========================================================================*
 *                              dump_core                                    *
 *===========================================================================*/
PRIVATE void dump_core(rmp)
register struct mproc *rmp;     /* whose core is to be dumped */
{
/* Make a core dump on the file "core", if possible. */

  int fd, fake_fd, nr_written, seg, slot;
  char *buf;
  vir_bytes current_sp;
  phys_bytes left;              /* careful; 64K might overflow vir_bytes */
  unsigned nr_to_write;         /* unsigned for arg to write() but < INT_MAX */
  long trace_data, trace_off;

  slot = (int) (rmp - mproc);

  /* Can core file be written?  We are operating in the user's FS environment,
   * so no special permission checks are needed.
   */
  if (rmp->mp_realuid != rmp->mp_effuid) return;
  if ( (fd = creat(core_name, CORE_MODE)) < 0) return;
  rmp->mp_sigstatus |= DUMPED;

  /* Make sure the stack segment is up to date.
   * We don't want adjust() to fail unless current_sp is preposterous,
   * but it might fail due to safety checking.  Also, we don't really want
   * the adjust() for sending a signal to fail due to safety checking.
   * Maybe make SAFETY_BYTES a parameter.
   */
  sys_getsp(slot, &current_sp);
  adjust(rmp, rmp->mp_seg[D].mem_len, current_sp);

  /* Write the memory map of all segments to begin the core file. */
  if (write(fd, (char *) rmp->mp_seg, (unsigned) sizeof rmp->mp_seg)
      != (unsigned) sizeof rmp->mp_seg) {
        close(fd);
        return;
  }

  /* Write out the whole kernel process table entry to get the regs. */
  trace_off = 0;
  while (sys_trace(3, slot, trace_off, &trace_data) == OK) {
        if (write(fd, (char *) &trace_data, (unsigned) sizeof (long))
            != (unsigned) sizeof (long)) {
                close(fd);
                return;
        }
        trace_off += sizeof (long);
  }

  /* Loop through segments and write the segments themselves out. */
  for (seg = 0; seg < NR_SEGS; seg++) {
        buf = (char *) ((vir_bytes) rmp->mp_seg[seg].mem_vir << CLICK_SHIFT);
        left = (phys_bytes) rmp->mp_seg[seg].mem_len << CLICK_SHIFT;
        fake_fd = (slot << 8) | (seg << 6) | fd;
```

```

        /* Loop through a segment, dumping it. */
        while (left != 0) {
                nr_to_write = (unsigned) MIN(left, DUMP_SIZE);
                if ( (nr_written = write(fake_fd, buf, nr_to_write)) < 0) {
                        close(fd);
                        return;
                }
                buf += nr_written;
                left -= nr_written;
        }
  }
  close(fd);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/getset.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file handles the 4 system calls that get and set uids and gids.
 * It also handles getpid(), setsid(), and getpgrp().  The code for each
 * one is so tiny that it hardly seemed worthwhile to make each a separate
 * function.
 */

#include "mm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/*===========================================================================*
 *                              do_getset                                    *
 *===========================================================================*/
PUBLIC int do_getset()
{
/* Handle GETUID, GETGID, GETPID, GETPGRP, SETUID, SETGID, SETSID.  The four
 * GETs and SETSID return their primary results in 'r'.  GETUID, GETGID, and
 * GETPID also return secondary results (the effective IDs, or the parent
 * process ID) in 'result2', which is returned to the user.
 */

  register struct mproc *rmp = mp;
  register int r;

  switch(mm_call) {
        case GETUID:
                r = rmp->mp_realuid;
                result2 = rmp->mp_effuid;
                break;

        case GETGID:
                r = rmp->mp_realgid;
                result2 = rmp->mp_effgid;
                break;

        case GETPID:
                r = mproc[who].mp_pid;
                result2 = mproc[rmp->mp_parent].mp_pid;
```

```
                break;

        case SETUID:
                if (rmp->mp_realuid != usr_id && rmp->mp_effuid != SUPER_USER)
                        return(EPERM);
                rmp->mp_realuid = usr_id;
                rmp->mp_effuid = usr_id;
                tell_fs(SETUID, who, usr_id, usr_id);
                r = OK;
                break;

        case SETGID:
                if (rmp->mp_realgid != grpid && rmp->mp_effuid != SUPER_USER)
                        return(EPERM);
                rmp->mp_realgid = grpid;
                rmp->mp_effgid = grpid;
                tell_fs(SETGID, who, grpid, grpid);
                r = OK;
                break;

        case SETSID:
                if (rmp->mp_procgrp == rmp->mp_pid) return(EPERM);
                rmp->mp_procgrp = rmp->mp_pid;
                tell_fs(SETSID, who, 0, 0);
                /*FALL THROUGH*/

        case GETPGRP:
                r = rmp->mp_procgrp;
                break;

        default:
                r = EINVAL;
                break;
  }
  return(r);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/trace.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file handles the memory manager's part of debugging, using the
 * ptrace system call. Most of the commands are passed on to the system
 * task for completion.
 *
 * The debugging commands available are:
 * T_STOP       stop the process
 * T_OK         enable tracing by parent for this process
 * T_GETINS     return value from instruction space
 * T_GETDATA    return value from data space
 * T_GETUSER    return value from user process table
 * T_SETINS     set value in instruction space
 * T_SETDATA    set value in data space
 * T_SETUSER    set value in user process table
 * T_RESUME     resume execution
 * T_EXIT       exit
```

```
 * T_STEP       set trace bit
 *
 * The T_OK and T_EXIT commands are handled here, and the T_RESUME and
 * T_STEP commands are partially handled here and completed by the system
 * task. The rest are handled entirely by the system task.
 */

#include "mm.h"
#include <sys/ptrace.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

#define NIL_MPROC       ((struct mproc *) 0)

FORWARD _PROTOTYPE( struct mproc *findproc, (pid_t lpid) );

/*===========================================================================*
 *                              do_trace                                     *
 *===========================================================================*/
PUBLIC int do_trace()
{
  register struct mproc *child;

  /* the T_OK call is made by the child fork of the debugger before it execs
   * the process to be traced
   */
  if (request == T_OK) {/* enable tracing by parent for this process */
        mp->mp_flags |= TRACED;
        mm_out.m2_l2 = 0;
        return(OK);
  }
  if ((child = findproc(pid)) == NIL_MPROC || !(child->mp_flags & STOPPED)) {
        return(ESRCH);
  }
  /* all the other calls are made by the parent fork of the debugger to
   * control execution of the child
   */
  switch (request) {
  case T_EXIT:          /* exit */
        mm_exit(child, (int)data);
        mm_out.m2_l2 = 0;
        return(OK);
  case T_RESUME:
  case T_STEP:          /* resume execution */
        if (data < 0 || data > _NSIG) return(EIO);
        if (data > 0) {         /* issue signal */
                child->mp_flags &= ~TRACED;  /* so signal is not diverted */
                sig_proc(child, (int) data);
                child->mp_flags |= TRACED;
        }
        child->mp_flags &= ~STOPPED;
        break;
  }
  if (sys_trace(request, (int) (child - mproc), taddr, &data) != OK)
        return(-errno);
  mm_out.m2_l2 = data;
  return(OK);
}
```

```
/*===========================================================================*
 *                              findproc                                     *
 *===========================================================================*/
PRIVATE struct mproc *findproc(lpid)
pid_t lpid;
{
  register struct mproc *rmp;

  for (rmp = &mproc[INIT_PROC_NR + 1]; rmp < &mproc[NR_PROCS]; rmp++)
        if (rmp->mp_flags & IN_USE && rmp->mp_pid == lpid) return(rmp);
  return(NIL_MPROC);
}

/*===========================================================================*
 *                              stop_proc                                    *
 *===========================================================================*/
PUBLIC void stop_proc(rmp, signo)
register struct mproc *rmp;
int signo;
{
/* A traced process got a signal so stop it. */

  register struct mproc *rpmp = mproc + rmp->mp_parent;

  if (sys_trace(-1, (int) (rmp - mproc), 0L, (long *) 0) != OK) return;
  rmp->mp_flags |= STOPPED;
  if (rpmp->mp_flags & WAITING) {
        rpmp->mp_flags &= ~WAITING;     /* parent is no longer waiting */
        reply(rmp->mp_parent, rmp->mp_pid, 0177 | (signo << 8), NIL_PTR);
  } else {
        rmp->mp_sigstatus = signo;
  }
  return;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/alloc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file is concerned with allocating and freeing arbitrary-size blocks of
 * physical memory on behalf of the FORK and EXEC system calls.  The key data
 * structure used is the hole table, which maintains a list of holes in memory.
 * It is kept sorted in order of increasing memory address. The addresses
 * it contains refer to physical memory, starting at absolute address 0
 * (i.e., they are not relative to the start of MM).  During system
 * initialization, that part of memory containing the interrupt vectors,
 * kernel, and MM are "allocated" to mark them as not available and to
 * remove them from the hole list.
 *
 * The entry points into this file are:
 *   alloc_mem: allocate a given sized chunk of memory
 *   free_mem:  release a previously allocated chunk of memory
 *   mem_init:  initialize the tables when MM start up
 *   max_hole:  returns the largest hole currently available
 */

#include "mm.h"
#include <minix/com.h>

```

```
#define NR_HOLES         128    /* max # entries in hole table */
#define NIL_HOLE (struct hole *) 0

PRIVATE struct hole {
  phys_clicks h_base;           /* where does the hole begin? */
  phys_clicks h_len;            /* how big is the hole? */
  struct hole *h_next;          /* pointer to next entry on the list */
} hole[NR_HOLES];


PRIVATE struct hole *hole_head; /* pointer to first hole */
PRIVATE struct hole *free_slots;        /* ptr to list of unused table slots */

FORWARD _PROTOTYPE( void del_slot, (struct hole *prev_ptr, struct hole *hp) );
FORWARD _PROTOTYPE( void merge, (struct hole *hp)                          );


/*===========================================================================*
 *                              alloc_mem                                    *
 *===========================================================================*/
PUBLIC phys_clicks alloc_mem(clicks)
phys_clicks clicks;             /* amount of memory requested */
{
/* Allocate a block of memory from the free list using first fit. The block
 * consists of a sequence of contiguous bytes, whose length in clicks is
 * given by 'clicks'.  A pointer to the block is returned.  The block is
 * always on a click boundary.  This procedure is called when memory is
 * needed for FORK or EXEC.
 */

  register struct hole *hp, *prev_ptr;
  phys_clicks old_base;

  hp = hole_head;
  while (hp != NIL_HOLE) {
        if (hp->h_len >= clicks) {
                /* We found a hole that is big enough.  Use it. */
                old_base = hp->h_base;  /* remember where it started */
                hp->h_base += clicks;   /* bite a piece off */
                hp->h_len -= clicks;    /* ditto */

                /* If hole is only partly used, reduce size and return. */
                if (hp->h_len != 0) return(old_base);

                /* The entire hole has been used up.  Manipulate free list. */
                del_slot(prev_ptr, hp);
                return(old_base);
        }

        prev_ptr = hp;
        hp = hp->h_next;
  }
  return(NO_MEM);
}


/*===========================================================================*
 *                              free_mem                                     *
 *===========================================================================*/
PUBLIC void free_mem(base, clicks)
```

```
phys_clicks base;               /* base address of block to free */
phys_clicks clicks;             /* number of clicks to free */
{
/* Return a block of free memory to the hole list.  The parameters tell where
 * the block starts in physical memory and how big it is.  The block is added
 * to the hole list.  If it is contiguous with an existing hole on either end,
 * it is merged with the hole or holes.
 */

  register struct hole *hp, *new_ptr, *prev_ptr;

  if (clicks == 0) return;
  if ( (new_ptr = free_slots) == NIL_HOLE) panic("Hole table full", NO_NUM);
  new_ptr->h_base = base;
  new_ptr->h_len = clicks;
  free_slots = new_ptr->h_next;
  hp = hole_head;

  /* If this block's address is numerically less than the lowest hole currently
   * available, or if no holes are currently available, put this hole on the
   * front of the hole list.
   */
  if (hp == NIL_HOLE || base <= hp->h_base) {
        /* Block to be freed goes on front of the hole list. */
        new_ptr->h_next = hp;
        hole_head = new_ptr;
        merge(new_ptr);
        return;
  }

  /* Block to be returned does not go on front of hole list. */
  while (hp != NIL_HOLE && base > hp->h_base) {
        prev_ptr = hp;
        hp = hp->h_next;
  }

  /* We found where it goes.  Insert block after 'prev_ptr'. */
  new_ptr->h_next = prev_ptr->h_next;
  prev_ptr->h_next = new_ptr;
  merge(prev_ptr);              /* sequence is 'prev_ptr', 'new_ptr', 'hp' */
}


/*===========================================================================*
 *                              del_slot                                     *
 *===========================================================================*/
PRIVATE void del_slot(prev_ptr, hp)
register struct hole *prev_ptr; /* pointer to hole entry just ahead of 'hp' */
register struct hole *hp;       /* pointer to hole entry to be removed */
{
/* Remove an entry from the hole list.  This procedure is called when a
 * request to allocate memory removes a hole in its entirety, thus reducing
 * the numbers of holes in memory, and requiring the elimination of one
 * entry in the hole list.
 */

  if (hp == hole_head)
        hole_head = hp->h_next;
  else
        prev_ptr->h_next = hp->h_next;
```

```
  hp->h_next = free_slots;
  free_slots = hp;
}


/*===========================================================================*
 *                              merge                                        *
 *===========================================================================*/
PRIVATE void merge(hp)
register struct hole *hp;       /* ptr to hole to merge with its successors */
{
/* Check for contiguous holes and merge any found.  Contiguous holes can occur
 * when a block of memory is freed, and it happens to abut another hole on
 * either or both ends.  The pointer 'hp' points to the first of a series of
 * three holes that can potentially all be merged together.
 */

  register struct hole *next_ptr;

  /* If 'hp' points to the last hole, no merging is possible.  If it does not,
   * try to absorb its successor into it and free the successor's table entry.
   */
  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
  if (hp->h_base + hp->h_len == next_ptr->h_base) {
        hp->h_len += next_ptr->h_len;   /* first one gets second one's mem */
        del_slot(hp, next_ptr);
  } else {
        hp = next_ptr;
  }

  /* If 'hp' now points to the last hole, return; otherwise, try to absorb its
   * successor into it.
   */
  if ( (next_ptr = hp->h_next) == NIL_HOLE) return;
  if (hp->h_base + hp->h_len == next_ptr->h_base) {
        hp->h_len += next_ptr->h_len;
        del_slot(hp, next_ptr);
  }
}


/*===========================================================================*
 *                              max_hole                                     *
 *===========================================================================*/
PUBLIC phys_clicks max_hole()
{
/* Scan the hole list and return the largest hole. */

  register struct hole *hp;
  register phys_clicks max;

  hp = hole_head;
  max = 0;
  while (hp != NIL_HOLE) {
        if (hp->h_len > max) max = hp->h_len;
        hp = hp->h_next;
  }
  return(max);
}
```

```
/*===========================================================================*
 *                              mem_init                                     *
 *===========================================================================*/
PUBLIC void mem_init(total, free)
phys_clicks *total, *free;              /* memory size summaries */
{
/* Initialize hole lists.  There are two lists: 'hole_head' points to a linked
 * list of all the holes (unused memory) in the system; 'free_slots' points to
 * a linked list of table entries that are not in use.  Initially, the former
 * list has one entry for each chunk of physical memory, and the second
 * list links together the remaining table slots.  As memory becomes more
 * fragmented in the course of time (i.e., the initial big holes break up into
 * smaller holes), new table slots are needed to represent them.  These slots
 * are taken from the list headed by 'free_slots'.
 */

  register struct hole *hp;
  phys_clicks base;             /* base address of chunk */
  phys_clicks size;             /* size of chunk */
  message mess;

  /* Put all holes on the free list. */
  for (hp = &hole[0]; hp < &hole[NR_HOLES]; hp++) hp->h_next = hp + 1;
  hole[NR_HOLES-1].h_next = NIL_HOLE;
  hole_head = NIL_HOLE;
  free_slots = &hole[0];

  /* Ask the kernel for chunks of physical memory and allocate a hole for
   * each of them.  The SYS_MEM call responds with the base and size of the
   * next chunk and the total amount of memory.
   */
  *free = 0;
  for (;;) {
        mess.m_type = SYS_MEM;
        if (sendrec(SYSTASK, &mess) != OK) panic("bad SYS_MEM?", NO_NUM);
        base = mess.m1_i1;
        size = mess.m1_i2;
        if (size == 0) break;           /* no more? */

        free_mem(base, size);
        *total = mess.m1_i3;
        *free += size;
  }
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/utility.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains some utility routines for MM.
 *
 * The entry points are:
 *   allowed:   see if an access is permitted
 *   no_sys:    this routine is called for invalid system call numbers
 *   panic:     MM has run aground of a fatal error and cannot continue
 *   tell_fs:   interface to FS
 */

#include "mm.h"
#include <sys/stat.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <fcntl.h>
#include <signal.h>             /* needed only because mproc.h needs it */
#include "mproc.h"

/*===========================================================================*
 *                              allowed                                      *
 *===========================================================================*/
PUBLIC int allowed(name_buf, s_buf, mask)
char *name_buf;                 /* pointer to file name to be EXECed */
struct stat *s_buf;             /* buffer for doing and returning stat struct*/
int mask;                       /* R_BIT, W_BIT, or X_BIT */
{
/* Check to see if file can be accessed.  Return EACCES or ENOENT if the access
 * is prohibited.  If it is legal open the file and return a file descriptor.
 */

  int fd;
  int save_errno;

  /* Use the fact that mask for access() is the same as the permissions mask.
   * E.g., X_BIT in <minix/const.h> is the same as X_OK in <unistd.h> and
   * S_IXOTH in <sys/stat.h>.  tell_fs(DO_CHDIR, ...) has set MM's real ids
   * to the user's effective ids, so access() works right for setuid programs.
   */
  if (access(name_buf, mask) < 0) return(-errno);

  /* The file is accessible but might not be readable.  Make it readable. */
  tell_fs(SETUID, MM_PROC_NR, (int) SUPER_USER, (int) SUPER_USER);

  /* Open the file and fstat it.  Restore the ids early to handle errors. */
  fd = open(name_buf, O_RDONLY);
  save_errno = errno;           /* open might fail, e.g. from ENFILE */
  tell_fs(SETUID, MM_PROC_NR, (int) mp->mp_effuid, (int) mp->mp_effuid);
  if (fd < 0) return(-save_errno);
  if (fstat(fd, s_buf) < 0) panic("allowed: fstat failed", NO_NUM);

  /* Only regular files can be executed. */
  if (mask == X_BIT && (s_buf->st_mode & I_TYPE) != I_REGULAR) {
        close(fd);
        return(EACCES);
  }
  return(fd);
```

```
}


/*===========================================================================*
 *                              no_sys                                       *
 *===========================================================================*/
PUBLIC int no_sys()
{
/* A system call number not implemented by MM has been requested. */

  return(EINVAL);
}


/*===========================================================================*
 *                              panic                                        *
 *===========================================================================*/
PUBLIC void panic(format, num)
char *format;                   /* format string */
int num;                        /* number to go with format string */
{
/* Something awful has happened.  Panics are caused when an internal
 * inconsistency is detected, e.g., a programming error or illegal value of a
 * defined constant.
 */

  printf("Memory manager panic: %s ", format);
  if (num != NO_NUM) printf("%d",num);
  printf("\n");
  tell_fs(SYNC, 0, 0, 0);       /* flush the cache to the disk */
  sys_abort(RBT_PANIC);
}


/*===========================================================================*
 *                              tell_fs                                      *
 *===========================================================================*/
PUBLIC void tell_fs(what, p1, p2, p3)
int what, p1, p2, p3;
{
/* This routine is only used by MM to inform FS of certain events:
 *      tell_fs(CHDIR, slot, dir, 0)
 *      tell_fs(EXEC, proc, 0, 0)
 *      tell_fs(EXIT, proc, 0, 0)
 *      tell_fs(FORK, parent, child, pid)
 *      tell_fs(SETGID, proc, realgid, effgid)
 *      tell_fs(SETSID, proc, 0, 0)
 *      tell_fs(SETUID, proc, realuid, effuid)
 *      tell_fs(SYNC, 0, 0, 0)
 *      tell_fs(UNPAUSE, proc, signr, 0)
 */

  message m;

  m.m1_i1 = p1;
  m.m1_i2 = p2;
  m.m1_i3 = p3;
  _taskcall(FS_PROC_NR, what, &m);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/mm/putk.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* MM must occasionally print some message.  It uses the standard library
 * routine printk().  (The name "printf" is really a macro defined as
 * "printk"). Printing is done by calling the TTY task directly, not going
 * through FS.
 */

#include "mm.h"
#include <minix/com.h>

#define BUF_SIZE          100   /* print buffer size */

PRIVATE int buf_count;          /* # characters in the buffer */
PRIVATE char print_buf[BUF_SIZE];       /* output is buffered here */
PRIVATE message putch_msg;      /* used for message to TTY task */

_PROTOTYPE( FORWARD void flush, (void) );

/*===========================================================================*
 *                              putk                                         *
 *===========================================================================*/
PUBLIC void putk(c)
int c;
{
/* Accumulate another character.  If 0 or buffer full, print it. */

  if (c == 0 || buf_count == BUF_SIZE) flush();
  if (c == '\n') putk('\r');
  if (c != 0) print_buf[buf_count++] = c;
}


/*===========================================================================*
 *                              flush                                        *
 *===========================================================================*/
PRIVATE void flush()
{
/* Flush the print buffer by calling TTY task. */

  if (buf_count == 0) return;
  putch_msg.m_type = DEV_WRITE;
  putch_msg.PROC_NR  = 0;
  putch_msg.TTY_LINE = 0;
  putch_msg.ADDRESS  = print_buf;
  putch_msg.COUNT = buf_count;
  sendrec(TTY, &putch_msg);
  buf_count = 0;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/fs.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This is the master header for fs.  It includes some other files
 * and defines the principal constants.
 */
#define _POSIX_SOURCE      1    /* tell headers to include POSIX stuff */
#define _MINIX             1    /* tell headers to include MINIX stuff */
#define _SYSTEM            1    /* tell headers that this is the kernel */

/* The following are so basic, all the *.c files get them automatically. */
#include <minix/config.h>       /* MUST be first */
#include <ansi.h>               /* MUST be second */
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>

#include <limits.h>
#include <errno.h>

#include <minix/syslib.h>

#include "const.h"
#include "type.h"
#include "proto.h"
#include "glo.h"
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/const.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Tables sizes */
#define V1_NR_DZONES       7    /* # direct zone numbers in a V1 inode */
#define V1_NR_TZONES       9    /* total # zone numbers in a V1 inode */
#define V2_NR_DZONES       7    /* # direct zone numbers in a V2 inode */
#define V2_NR_TZONES      10    /* total # zone numbers in a V2 inode */

#define NR_FILPS         128    /* # slots in filp table */
#define NR_INODES         64    /* # slots in "in core" inode table */
#define NR_SUPERS          8    /* # slots in super block table */
#define NR_LOCKS           8    /* # slots in the file locking table */

/* The type of sizeof may be (unsigned) long.  Use the following macro for
 * taking the sizes of small objects so that there are no surprises like
 * (small) long constants being passed to routines expecting an int.
 */
#define usizeof(t) ((unsigned) sizeof(t))

/* File system types. */
#define SUPER_MAGIC   0x137F    /* magic number contained in super-block */
#define SUPER_REV     0x7F13    /* magic # when 68000 disk read on PC or vv */
#define SUPER_V2      0x2468    /* magic # for V2 file systems */
#define SUPER_V2_REV  0x6824    /* V2 magic written on PC, read on 68K or vv */

#define V1                 1    /* version number of V1 file systems */
#define V2                 2    /* version number of V2 file systems */
```

```

/* Miscellaneous constants */
#define SU_UID   ((uid_t) 0)    /* super_user's uid_t */
#define SYS_UID  ((uid_t) 0)    /* uid_t for processes MM and INIT */
#define SYS_GID  ((gid_t) 0)    /* gid_t for processes MM and INIT */
#define NORMAL            0     /* forces get_block to do disk read */
#define NO_READ           1     /* prevents get_block from doing disk read */
#define PREFETCH          2     /* tells get_block not to read or mark dev */

#define XPIPE  (-NR_TASKS-1)    /* used in fp_task when susp'd on pipe */
#define XOPEN  (-NR_TASKS-2)    /* used in fp_task when susp'd on open */
#define XLOCK  (-NR_TASKS-3)    /* used in fp_task when susp'd on lock */
#define XPOPEN (-NR_TASKS-4)    /* used in fp_task when susp'd on pipe open */

#define NO_BIT   ((bit_t) 0)    /* returned by alloc_bit() to signal failure */

#define DUP_MASK       0100     /* mask to distinguish dup2 from dup */

#define LOOK_UP           0     /* tells search_dir to lookup string */
#define ENTER             1     /* tells search_dir to make dir entry */
#define DELETE            2     /* tells search_dir to delete entry */
#define IS_EMPTY          3     /* tells search_dir to ret. OK or ENOTEMPTY */

#define CLEAN             0     /* disk and memory copies identical */
#define DIRTY             1     /* disk and memory copies differ */
#define ATIME           002     /* set if atime field needs updating */
#define CTIME           004     /* set if ctime field needs updating */
#define MTIME           010     /* set if mtime field needs updating */

#define BYTE_SWAP         0     /* tells conv2/conv4 to swap bytes */
#define DONT_SWAP         1     /* tells conv2/conv4 not to swap bytes */

#define END_OF_FILE   (-104)    /* eof detected */

#define ROOT_INODE        1     /* inode number for root directory */
#define BOOT_BLOCK  ((block_t) 0)       /* block number of boot block */
#define SUPER_BLOCK ((block_t) 1)       /* block number of super block */

#define DIR_ENTRY_SIZE       usizeof (struct direct)  /* # bytes/dir entry   */
#define NR_DIR_ENTRIES   (BLOCK_SIZE/DIR_ENTRY_SIZE)  /* # dir entries/blk   */
#define SUPER_SIZE      usizeof (struct super_block)  /* super_block size    */
#define PIPE_SIZE          (V1_NR_DZONES*BLOCK_SIZE)  /* pipe size in bytes  */
#define BITMAP_CHUNKS (BLOCK_SIZE/usizeof (bitchunk_t))/* # map chunks/blk   */

/* Derived sizes pertaining to the V1 file system. */
#define V1_ZONE_NUM_SIZE           usizeof (zone1_t)  /* # bytes in V1 zone  */
#define V1_INODE_SIZE             usizeof (d1_inode)  /* bytes in V1 dsk ino */
#define V1_INDIRECTS   (BLOCK_SIZE/V1_ZONE_NUM_SIZE)  /* # zones/indir block */
#define V1_INODES_PER_BLOCK (BLOCK_SIZE/V1_INODE_SIZE)/* # V1 dsk inodes/blk */

/* Derived sizes pertaining to the V2 file system. */
#define V2_ZONE_NUM_SIZE            usizeof (zone_t)  /* # bytes in V2 zone  */
#define V2_INODE_SIZE             usizeof (d2_inode)  /* bytes in V2 dsk ino */
#define V2_INDIRECTS   (BLOCK_SIZE/V2_ZONE_NUM_SIZE)  /* # zones/indir block */
#define V2_INODES_PER_BLOCK (BLOCK_SIZE/V2_INODE_SIZE)/* # V2 dsk inodes/blk */

#define printf printk
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/type.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Declaration of the V1 inode as it is on the disk (not in core). */
typedef struct {                /* V1.x disk inode */
  mode_t d1_mode;               /* file type, protection, etc. */
  uid_t d1_uid;                 /* user id of the file's owner */
  off_t d1_size;                /* current file size in bytes */
  time_t d1_mtime;              /* when was file data last changed */
  gid_t d1_gid;                 /* group number */
  nlink_t d1_nlinks;            /* how many links to this file */
  u16_t d1_zone[V1_NR_TZONES];  /* block nums for direct, ind, and dbl ind */
} d1_inode;

/* Declaration of the V2 inode as it is on the disk (not in core). */
typedef struct {                /* V2.x disk inode */
  mode_t d2_mode;               /* file type, protection, etc. */
  u16_t d2_nlinks;              /* how many links to this file. HACK! */
  uid_t d2_uid;                 /* user id of the file's owner. */
  u16_t d2_gid;                 /* group number HACK! */
  off_t d2_size;                /* current file size in bytes */
  time_t d2_atime;              /* when was file data last accessed */
  time_t d2_mtime;              /* when was file data last changed */
  time_t d2_ctime;              /* when was inode data last changed */
  zone_t d2_zone[V2_NR_TZONES]; /* block nums for direct, ind, and dbl ind */
} d2_inode;
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/proto.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Function prototypes. */

/* Structs used in prototypes must be declared as such first. */
struct buf;
struct filp;
struct inode;
struct super_block;

/* cache.c */
_PROTOTYPE( zone_t alloc_zone, (Dev_t dev, zone_t z)                    );
_PROTOTYPE( void flushall, (Dev_t dev)                                  );
_PROTOTYPE( void free_zone, (Dev_t dev, zone_t numb)                    );
_PROTOTYPE( struct buf *get_block, (Dev_t dev, block_t block,int only_search));
_PROTOTYPE( void invalidate, (Dev_t device)                             );
_PROTOTYPE( void put_block, (struct buf *bp, int block_type)            );
_PROTOTYPE( void rw_block, (struct buf *bp, int rw_flag)                );
_PROTOTYPE( void rw_scattered, (Dev_t dev,
                        struct buf **bufq, int bufqsize, int rw_flag)   );

/* device.c */
_PROTOTYPE( void call_task, (int task_nr, message *mess_ptr)            );
_PROTOTYPE( void dev_opcl, (int task_nr, message *mess_ptr)             );
_PROTOTYPE( int dev_io, (int rw_flag, int nonblock, Dev_t dev,
                        off_t pos, int bytes, int proc, char *buff)     );
_PROTOTYPE( int do_ioctl, (void)                                        );
```

```
_PROTOTYPE( void no_dev, (int task_nr, message *m_ptr)                  );
_PROTOTYPE( void call_ctty, (int task_nr, message *mess_ptr)            );
_PROTOTYPE( void tty_open, (int task_nr, message *mess_ptr)             );
_PROTOTYPE( void ctty_close, (int task_nr, message *mess_ptr)           );
_PROTOTYPE( void ctty_open, (int task_nr, message *mess_ptr)            );
_PROTOTYPE( int do_setsid, (void)                                       );
#if ENABLE_NETWORKING
_PROTOTYPE( void net_open, (int task_nr, message *mess_ptr)             );
#else
#define net_open  0
#endif

/* filedes.c */
_PROTOTYPE( struct filp *find_filp, (struct inode *rip, Mode_t bits)    );
_PROTOTYPE( int get_fd, (int start, Mode_t bits, int *k, struct filp **fpt) );
_PROTOTYPE( struct filp *get_filp, (int fild)                           );

/* inode.c */
_PROTOTYPE( struct inode *alloc_inode, (Dev_t dev, Mode_t bits)         );
_PROTOTYPE( void dup_inode, (struct inode *ip)                          );
_PROTOTYPE( void free_inode, (Dev_t dev, Ino_t numb)                    );
_PROTOTYPE( struct inode *get_inode, (Dev_t dev, int numb)              );
_PROTOTYPE( void put_inode, (struct inode *rip)                         );
_PROTOTYPE( void update_times, (struct inode *rip)                      );
_PROTOTYPE( void rw_inode, (struct inode *rip, int rw_flag)             );
_PROTOTYPE( void wipe_inode, (struct inode *rip)                        );

/* link.c */
_PROTOTYPE( int do_link, (void)                                         );
_PROTOTYPE( int do_unlink, (void)                                       );
_PROTOTYPE( int do_rename, (void)                                       );
_PROTOTYPE( void truncate, (struct inode *rip)                          );

/* lock.c */
_PROTOTYPE( int lock_op, (struct filp *f, int req)                      );
_PROTOTYPE( void lock_revive, (void)                                    );

/* main.c */
_PROTOTYPE( void main, (void)                                           );
_PROTOTYPE( void reply, (int whom, int result)                          );

/* misc.c */
_PROTOTYPE( int do_dup, (void)                                          );
_PROTOTYPE( int do_exit, (void)                                         );
_PROTOTYPE( int do_fcntl, (void)                                        );
_PROTOTYPE( int do_fork, (void)                                         );
_PROTOTYPE( int do_exec, (void)                                         );
_PROTOTYPE( int do_revive, (void)                                       );
_PROTOTYPE( int do_set, (void)                                          );
_PROTOTYPE( int do_sync, (void)                                         );

/* mount.c */
_PROTOTYPE( int do_mount, (void)                                        );
_PROTOTYPE( int do_umount, (void)                                       );

/* open.c */
_PROTOTYPE( int do_close, (void)                                        );
_PROTOTYPE( int do_creat, (void)                                        );
_PROTOTYPE( int do_lseek, (void)                                        );
_PROTOTYPE( int do_mknod, (void)                                        );
```

```
_PROTOTYPE( int do_mkdir, (void)                                        );
_PROTOTYPE( int do_open, (void)                                         );

/* path.c */
_PROTOTYPE( struct inode *advance,(struct inode *dirp, char string[NAME_MAX]));
_PROTOTYPE( int search_dir, (struct inode *ldir_ptr,
                        char string [NAME_MAX], ino_t *numb, int flag)  );
_PROTOTYPE( struct inode *eat_path, (char *path)                        );
_PROTOTYPE( struct inode *last_dir, (char *path, char string [NAME_MAX]));

/* pipe.c */
_PROTOTYPE( int do_pipe, (void)                                         );
_PROTOTYPE( int do_unpause, (void)                                      );
_PROTOTYPE( int pipe_check, (struct inode *rip, int rw_flag,
                        int oflags, int bytes, off_t position, int *canwrite));
_PROTOTYPE( void release, (struct inode *ip, int call_nr, int count)    );
_PROTOTYPE( void revive, (int proc_nr, int bytes)                       );
_PROTOTYPE( void suspend, (int task)                                    );

/* protect.c */
_PROTOTYPE( int do_access, (void)                                       );
_PROTOTYPE( int do_chmod, (void)                                        );
_PROTOTYPE( int do_chown, (void)                                        );
_PROTOTYPE( int do_umask, (void)                                        );
_PROTOTYPE( int forbidden, (struct inode *rip, Mode_t access_desired)   );
_PROTOTYPE( int read_only, (struct inode *ip)                           );

/* putk.c */
_PROTOTYPE( void putk, (int c)                                          );

/* read.c */
_PROTOTYPE( int do_read, (void)                                         );
_PROTOTYPE( struct buf *rahead, (struct inode *rip, block_t baseblock,
                        off_t position, unsigned bytes_ahead)           );
_PROTOTYPE( void read_ahead, (void)                                     );
_PROTOTYPE( block_t read_map, (struct inode *rip, off_t position)       );
_PROTOTYPE( int read_write, (int rw_flag)                               );
_PROTOTYPE( zone_t rd_indir, (struct buf *bp, int index)                );

/* stadir.c */
_PROTOTYPE( int do_chdir, (void)                                        );
_PROTOTYPE( int do_chroot, (void)                                       );
_PROTOTYPE( int do_fstat, (void)                                        );
_PROTOTYPE( int do_stat, (void)                                         );

/* super.c */
_PROTOTYPE( bit_t alloc_bit, (struct super_block *sp, int map, bit_t origin));
_PROTOTYPE( void free_bit, (struct super_block *sp, int map,
                                                bit_t bit_returned)     );
_PROTOTYPE( struct super_block *get_super, (Dev_t dev)                  );
_PROTOTYPE( int mounted, (struct inode *rip)                            );
_PROTOTYPE( int read_super, (struct super_block *sp)                    );

/* time.c */
_PROTOTYPE( int do_stime, (void)                                        );
_PROTOTYPE( int do_time, (void)                                         );
_PROTOTYPE( int do_tims, (void)                                         );
_PROTOTYPE( int do_utime, (void)                                        );

/* utility.c */
```

```
_PROTOTYPE( time_t clock_time, (void)                                   );
_PROTOTYPE( unsigned conv2, (int norm, int w)                           );
_PROTOTYPE( long conv4, (int norm, long x)                              );
_PROTOTYPE( int fetch_name, (char *path, int len, int flag)             );
_PROTOTYPE( int no_sys, (void)                                          );
_PROTOTYPE( void panic, (char *format, int num)                         );

/* write.c */
_PROTOTYPE( void clear_zone, (struct inode *rip, off_t pos, int flag)   );
_PROTOTYPE( int do_write, (void)                                        );
_PROTOTYPE( struct buf *new_block, (struct inode *rip, off_t position)  );
_PROTOTYPE( void zero_block, (struct buf *bp)                           );
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/glo.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* EXTERN should be extern except for the table file */
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif

/* File System global variables */
EXTERN struct fproc *fp;        /* pointer to caller's fproc struct */
EXTERN int super_user;          /* 1 if caller is super_user, else 0 */
EXTERN int dont_reply;          /* normally 0; set to 1 to inhibit reply */
EXTERN int susp_count;          /* number of procs suspended on pipe */
EXTERN int nr_locks;            /* number of locks currently in place */
EXTERN int reviving;            /* number of pipe processes to be revived */
EXTERN off_t rdahedpos;         /* position to read ahead */
EXTERN struct inode *rdahed_inode;      /* pointer to inode to read ahead */

/* The parameters of the call are kept here. */
EXTERN message m;               /* the input message itself */
EXTERN message m1;              /* the output message used for reply */
EXTERN int who;                 /* caller's proc number */
EXTERN int fs_call;             /* system call number */
EXTERN char user_path[PATH_MAX];/* storage for user path name */

/* The following variables are used for returning results to the caller. */
EXTERN int err_code;            /* temporary storage for error number */
EXTERN int rdwt_err;            /* status of last disk i/o request */

/* Data which need initialization. */
extern _PROTOTYPE (int (*call_vector[]), (void) ); /* sys call table */
extern int max_major;  /* maximum major device (+ 1) */
extern char dot1[2];   /* dot1 (&dot1[0]) and dot2 (&dot2[0]) have a special */
extern char dot2[3];   /* meaning to search_dir: no access permission check. */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/fproc.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This is the per-process information.  A slot is reserved for each potential
 * process. Thus NR_PROCS must be the same as in the kernel. It is not possible
 * or even necessary to tell when a slot is free here.
 */


EXTERN struct fproc {
  mode_t fp_umask;              /* mask set by umask system call */
  struct inode *fp_workdir;     /* pointer to working directory's inode */
  struct inode *fp_rootdir;     /* pointer to current root dir (see chroot) */
  struct filp *fp_filp[OPEN_MAX];/* the file descriptor table */
  uid_t fp_realuid;             /* real user id */
  uid_t fp_effuid;              /* effective user id */
  gid_t fp_realgid;             /* real group id */
  gid_t fp_effgid;              /* effective group id */
  dev_t fp_tty;                 /* major/minor of controlling tty */
  int fp_fd;                    /* place to save fd if rd/wr can't finish */
  char *fp_buffer;              /* place to save buffer if rd/wr can't finish*/
  int  fp_nbytes;               /* place to save bytes if rd/wr can't finish */
  int  fp_cum_io_partial;       /* partial byte count if rd/wr can't finish */
  char fp_suspended;            /* set to indicate process hanging */
  char fp_revived;              /* set to indicate process being revived */
  char fp_task;                 /* which task is proc suspended on */
  char fp_sesldr;               /* true if proc is a session leader */
  pid_t fp_pid;                 /* process id */
  long fp_cloexec;              /* bit map for POSIX Table 6-2 FD_CLOEXEC */
} fproc[NR_PROCS];

/* Field values. */
#define NOT_SUSPENDED      0    /* process is not suspended on pipe or task */
#define SUSPENDED          1    /* process is suspended on pipe or task */
#define NOT_REVIVING       0    /* process is not being revived */
#define REVIVING           1    /* process is being revived from suspension */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/buf.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* Buffer (block) cache.  To acquire a block, a routine calls get_block(),
 * telling which block it wants.  The block is then regarded as "in use"
 * and has its 'b_count' field incremented.  All the blocks that are not
 * in use are chained together in an LRU list, with 'front' pointing
 * to the least recently used block, and 'rear' to the most recently used
 * block.  A reverse chain, using the field b_prev is also maintained.
 * Usage for LRU is measured by the time the put_block() is done.  The second
 * parameter to put_block() can violate the LRU order and put a block on the
 * front of the list, if it will probably not be needed soon.  If a block
 * is modified, the modifying routine must set b_dirt to DIRTY, so the block
 * will eventually be rewritten to the disk.
 */

#include <sys/dir.h>                    /* need struct direct */

```

```
EXTERN struct buf {
  /* Data portion of the buffer. */
  union {
    char b__data[BLOCK_SIZE];                /* ordinary user data */
    struct direct b__dir[NR_DIR_ENTRIES];    /* directory block */
    zone1_t b__v1_ind[V1_INDIRECTS];         /* V1 indirect block */
    zone_t  b__v2_ind[V2_INDIRECTS];         /* V2 indirect block */
    d1_inode b__v1_ino[V1_INODES_PER_BLOCK]; /* V1 inode block */
    d2_inode b__v2_ino[V2_INODES_PER_BLOCK]; /* V2 inode block */
    bitchunk_t b__bitmap[BITMAP_CHUNKS];     /* bit map block */
  } b;

  /* Header portion of the buffer. */
  struct buf *b_next;           /* used to link all free bufs in a chain */
  struct buf *b_prev;           /* used to link all free bufs the other way */
  struct buf *b_hash;           /* used to link bufs on hash chains */
  block_t b_blocknr;            /* block number of its (minor) device */
  dev_t b_dev;                  /* major | minor device where block resides */
  char b_dirt;                  /* CLEAN or DIRTY */
  char b_count;                 /* number of users of this buffer */
} buf[NR_BUFS];

/* A block is free if b_dev == NO_DEV. */

#define NIL_BUF ((struct buf *) 0)      /* indicates absence of a buffer */

/* These defs make it possible to use to bp->b_data instead of bp->b.b__data */
#define b_data   b.b__data
#define b_dir    b.b__dir
#define b_v1_ind b.b__v1_ind
#define b_v2_ind b.b__v2_ind
#define b_v1_ino b.b__v1_ino
#define b_v2_ino b.b__v2_ino
#define b_bitmap b.b__bitmap

EXTERN struct buf *buf_hash[NR_BUF_HASH];       /* the buffer hash table */

EXTERN struct buf *front;       /* points to least recently used free block */
EXTERN struct buf *rear;        /* points to most recently used free block */
EXTERN int bufs_in_use;         /* # bufs currently in use (not on free list)*/

/* When a block is released, the type of usage is passed to put_block(). */
#define WRITE_IMMED        0100 /* block should be written to disk now */
#define ONE_SHOT           0200 /* set if block not likely to be needed soon */

#define INODE_BLOCK       (0 + MAYBE_WRITE_IMMED)        /* inode block */
#define DIRECTORY_BLOCK   (1 + MAYBE_WRITE_IMMED)        /* directory block */
#define INDIRECT_BLOCK    (2 + MAYBE_WRITE_IMMED)        /* pointer block */
#define MAP_BLOCK         (3 + MAYBE_WRITE_IMMED)        /* bit map */
#define ZUPER_BLOCK       (4 + WRITE_IMMED + ONE_SHOT)   /* super block */
#define FULL_DATA_BLOCK    5                             /* data, fully used */
#define PARTIAL_DATA_BLOCK 6                             /* data, partly used*/

#define HASH_MASK (NR_BUF_HASH - 1)     /* mask for hashing block numbers */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/dev.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Device table.  This table is indexed by major device number.  It provides
 * the link between major device numbers and the routines that process them.
 */

typedef _PROTOTYPE (void (*dmap_t), (int task, message *m_ptr) );

extern struct dmap {
  dmap_t dmap_open;
  dmap_t dmap_rw;
  dmap_t dmap_close;
  int dmap_task;
} dmap[];

```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/file.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This is the filp table.  It is an intermediary between file descriptors and
 * inodes.  A slot is free if filp_count == 0.
 */

EXTERN struct filp {
  mode_t filp_mode;             /* RW bits, telling how file is opened */
  int filp_flags;               /* flags from open and fcntl */
  int filp_count;               /* how many file descriptors share this slot?*/
  struct inode *filp_ino;       /* pointer to the inode */
  off_t filp_pos;               /* file position */
} filp[NR_FILPS];

#define FILP_CLOSED     0       /* filp_mode: associated device closed */

#define NIL_FILP (struct filp *) 0      /* indicates absence of a filp slot */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/lock.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This is the file locking table.  Like the filp table, it points to the
 * inode table, however, in this case to achieve advisory locking.
 */
EXTERN struct file_lock {
  short lock_type;              /* F_RDLOCK or F_WRLOCK; 0 means unused slot */
  pid_t lock_pid;               /* pid of the process holding the lock */
  struct inode *lock_inode;     /* pointer to the inode locked */
  off_t lock_first;             /* offset of first byte locked */
  off_t lock_last;              /* offset of last byte locked */
} file_lock[NR_LOCKS];
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/inode.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```c
/* Inode table.  This table holds inodes that are currently in use.  In some
 * cases they have been opened by an open() or creat() system call, in other
 * cases the file system itself needs the inode for one reason or another,
 * such as to search a directory for a path name.
 * The first part of the struct holds fields that are present on the
 * disk; the second part holds fields not present on the disk.
 * The disk inode part is also declared in "type.h" as 'd1_inode' for V1
 * file systems and 'd2_inode' for V2 file systems.
 */

EXTERN struct inode {
  mode_t i_mode;                /* file type, protection, etc. */
  nlink_t i_nlinks;             /* how many links to this file */
  uid_t i_uid;                  /* user id of the file's owner */
  gid_t i_gid;                  /* group number */
  off_t i_size;                 /* current file size in bytes */
  time_t i_atime;               /* time of last access (V2 only) */
  time_t i_mtime;               /* when was file data last changed */
  time_t i_ctime;               /* when was inode itself changed (V2 only)*/
  zone_t i_zone[V2_NR_TZONES];  /* zone numbers for direct, ind, and dbl ind */

  /* The following items are not present on the disk. */
  dev_t i_dev;                  /* which device is the inode on */
  ino_t i_num;                  /* inode number on its (minor) device */
  int i_count;                  /* # times inode used; 0 means slot is free */
  int i_ndzones;                /* # direct zones (Vx_NR_DZONES) */
  int i_nindirs;                /* # indirect zones per indirect block */
  struct super_block *i_sp;     /* pointer to super block for inode's device */
  char i_dirt;                  /* CLEAN or DIRTY */
  char i_pipe;                  /* set to I_PIPE if pipe */
  char i_mount;                 /* this bit is set if file mounted on */
  char i_seek;                  /* set on LSEEK, cleared on READ/WRITE */
  char i_update;                /* the ATIME, CTIME, and MTIME bits are here */
} inode[NR_INODES];


#define NIL_INODE (struct inode *) 0    /* indicates absence of inode slot */

/* Field values.  Note that CLEAN and DIRTY are defined in "const.h" */
#define NO_PIPE            0    /* i_pipe is NO_PIPE if inode is not a pipe */
#define I_PIPE             1    /* i_pipe is I_PIPE if inode is a pipe */
#define NO_MOUNT           0    /* i_mount is NO_MOUNT if file not mounted on*/
#define I_MOUNT            1    /* i_mount is I_MOUNT if file mounted on */
#define NO_SEEK            0    /* i_seek = NO_SEEK if last op was not SEEK */
#define ISEEK              1    /* i_seek = ISEEK if last op was SEEK */
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/param.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* The following names are synonyms for the variables in the input message. */
#define acc_time      m.m2_l1
#define addr          m.m1_i3
#define buffer        m.m1_p1
#define child         m.m1_i2
#define co_mode       m.m1_i1
#define eff_grp_id    m.m1_i3
#define eff_user_id   m.m1_i3
#define erki          m.m1_p1
#define fd            m.m1_i1
#define fd2           m.m1_i2
#define ioflags       m.m1_i3
#define group         m.m1_i3
#define real_grp_id   m.m1_i2
#define ls_fd         m.m2_i1
#define mk_mode       m.m1_i2
#define mode          m.m3_i2
#define c_mode        m.m1_i3
#define c_name        m.m1_p1
#define name          m.m3_p1
#define name1         m.m1_p1
#define name2         m.m1_p2
#define name_length   m.m3_i1
#define name1_length  m.m1_i1
#define name2_length  m.m1_i2
#define nbytes        m.m1_i2
#define offset        m.m2_l1
#define owner         m.m1_i2
#define parent        m.m1_i1
#define pathname      m.m3_ca1
#define pid           m.m1_i3
#define pro           m.m1_i1
#define rd_only       m.m1_i3
#define real_user_id  m.m1_i2
#define request       m.m1_i2
#define sig           m.m1_i2
#define slot1         m.m1_i1
#define tp            m.m2_l1
#define utime_actime  m.m2_l1
#define utime_modtime m.m2_l2
#define utime_file    m.m2_p1
#define utime_length  m.m2_i1
#define whence        m.m2_i2

/* The following names are synonyms for the variables in the output message. */
#define reply_type    m1.m_type
#define reply_l1      m1.m2_l1
#define reply_i1      m1.m1_i1
#define reply_i2      m1.m1_i2
#define reply_t1      m1.m4_l1
#define reply_t2      m1.m4_l2
#define reply_t3      m1.m4_l3
#define reply_t4      m1.m4_l4
#define reply_t5      m1.m4_l5
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/super.h
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* Super block table.  The root file system and every mounted file system
 * has an entry here.  The entry holds information about the sizes of the bit
 * maps and inodes.  The s_ninodes field gives the number of inodes available
 * for files and directories, including the root directory.  Inode 0 is
 * on the disk, but not used.  Thus s_ninodes = 4 means that 5 bits will be
 * used in the bit map, bit 0, which is always 1 and not used, and bits 1-4
 * for files and directories.  The disk layout is:
 *
 *      Item        # blocks
 *    boot block      1
 *    super block     1
 *    inode map     s_imap_blocks
 *    zone map      s_zmap_blocks
 *    inodes        (s_ninodes + 'inodes per block' - 1)/'inodes per block'
 *    unused        whatever is needed to fill out the current zone
 *    data zones    (s_zones - s_firstdatazone) << s_log_zone_size
 *
 * A super_block slot is free if s_dev == NO_DEV.
 */


EXTERN struct super_block {
  ino_t s_ninodes;              /* # usable inodes on the minor device */
  zone1_t  s_nzones;            /* total device size, including bit maps etc */
  short s_imap_blocks;          /* # of blocks used by inode bit map */
  short s_zmap_blocks;          /* # of blocks used by zone bit map */
  zone1_t s_firstdatazone;      /* number of first data zone */
  short s_log_zone_size;        /* log2 of blocks/zone */
  off_t s_max_size;             /* maximum file size on this device */
  short s_magic;                /* magic number to recognize super-blocks */
  short s_pad;                  /* try to avoid compiler-dependent padding */
  zone_t s_zones;               /* number of zones (replaces s_nzones in V2) */

  /* The following items are only used when the super_block is in memory. */
  struct inode *s_isup;         /* inode for root dir of mounted file sys */
  struct inode *s_imount;       /* inode mounted on */
  unsigned s_inodes_per_block;  /* precalculated from magic number */
  dev_t s_dev;                  /* whose super block is this? */
  int s_rd_only;                /* set to 1 iff file sys mounted read only */
  int s_native;                 /* set to 1 iff not byte swapped file system */
  int s_version;                /* file system version, zero means bad magic */
  int s_ndzones;                /* # direct zones in an inode */
  int s_nindirs;                /* # indirect zones per indirect block */
  bit_t s_isearch;              /* inodes below this bit number are in use */
  bit_t s_zsearch;              /* all zones below this bit number are in use*/
} super_block[NR_SUPERS];

#define NIL_SUPER (struct super_block *) 0
#define IMAP            0       /* operating on the inode bit map */
#define ZMAP            1       /* operating on the zone bit map */
```

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/table.c
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#define _TABLE

#include "fs.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "super.h"

PUBLIC _PROTOTYPE (int (*call_vector[NCALLS]), (void) ) = {
        no_sys,         /*  0 = unused  */
        do_exit,        /*  1 = exit    */
        do_fork,        /*  2 = fork    */
        do_read,        /*  3 = read    */
        do_write,       /*  4 = write   */
        do_open,        /*  5 = open    */
        do_close,       /*  6 = close   */
        no_sys,         /*  7 = wait    */
        do_creat,       /*  8 = creat   */
        do_link,        /*  9 = link    */
        do_unlink,      /* 10 = unlink  */
        no_sys,         /* 11 = waitpid */
        do_chdir,       /* 12 = chdir   */
        do_time,        /* 13 = time    */
        do_mknod,       /* 14 = mknod   */
        do_chmod,       /* 15 = chmod   */
        do_chown,       /* 16 = chown   */
        no_sys,         /* 17 = break   */
        do_stat,        /* 18 = stat    */
        do_lseek,       /* 19 = lseek   */
        no_sys,         /* 20 = getpid  */
        do_mount,       /* 21 = mount   */
        do_umount,      /* 22 = umount  */
        do_set,         /* 23 = setuid  */
        no_sys,         /* 24 = getuid  */
        do_stime,       /* 25 = stime   */
        no_sys,         /* 26 = ptrace  */
        no_sys,         /* 27 = alarm   */
        do_fstat,       /* 28 = fstat   */
        no_sys,         /* 29 = pause   */
        do_utime,       /* 30 = utime   */
        no_sys,         /* 31 = (stty)  */
        no_sys,         /* 32 = (gtty)  */
        do_access,      /* 33 = access  */
        no_sys,         /* 34 = (nice)  */
        no_sys,         /* 35 = (ftime) */
        do_sync,        /* 36 = sync    */
```

```c
        no_sys,         /* 37 = kill    */
        do_rename,      /* 38 = rename  */
        do_mkdir,       /* 39 = mkdir   */
        do_unlink,      /* 40 = rmdir   */
        do_dup,         /* 41 = dup     */
        do_pipe,        /* 42 = pipe    */
        do_tims,        /* 43 = times   */
        no_sys,         /* 44 = (prof)  */
        no_sys,         /* 45 = unused  */
        do_set,         /* 46 = setgid  */
        no_sys,         /* 47 = getgid  */
        no_sys,         /* 48 = (signal)*/
        no_sys,         /* 49 = unused  */
        no_sys,         /* 50 = unused  */
        no_sys,         /* 51 = (acct)  */
        no_sys,         /* 52 = (phys)  */
        no_sys,         /* 53 = (lock)  */
        do_ioctl,       /* 54 = ioctl   */
        do_fcntl,       /* 55 = fcntl   */
        no_sys,         /* 56 = (mpx)   */
        no_sys,         /* 57 = unused  */
        no_sys,         /* 58 = unused  */
        do_exec,        /* 59 = execve  */
        do_umask,       /* 60 = umask   */
        do_chroot,      /* 61 = chroot  */
        do_setsid,      /* 62 = setsid  */
        no_sys,         /* 63 = getpgrp */

        no_sys,         /* 64 = KSIG: signals originating in the kernel */
        do_unpause,     /* 65 = UNPAUSE */
        no_sys,         /* 66 = unused  */
        do_revive,      /* 67 = REVIVE  */
        no_sys,         /* 68 = TASK_REPLY      */
        no_sys,         /* 69 = unused */
        no_sys,         /* 70 = unused */
        no_sys,         /* 71 = SIGACTION */
        no_sys,         /* 72 = SIGSUSPEND */
        no_sys,         /* 73 = SIGPENDING */
        no_sys,         /* 74 = SIGPROCMASK */
        no_sys,         /* 75 = SIGRETURN */
        no_sys,         /* 76 = REBOOT */
};


/* Some devices may or may not be there in the next table. */
#define DT(enable, open, rw, close, task) \
        { (enable ? (open) : no_dev), (enable ? (rw) : no_dev), \
          (enable ? (close) : no_dev), (enable ? (task) : 0) },

/* The order of the entries here determines the mapping between major device
 * numbers and tasks.  The first entry (major device 0) is not used.  The
 * next entry is major device 1, etc.  Character and block devices can be
 * intermixed at random.  If this ordering is changed, the devices in
 * <include/minix/boot.h> must be changed to correspond to the new values.
 * Note that the major device numbers used in /dev are NOT the same as the
 * task numbers used inside the kernel (as defined in <include/minix/com.h>).
 * Also note that if /dev/mem is changed from 1, NULL_MAJOR must be changed
 * in <include/minix/com.h>.
 */
PUBLIC struct dmap dmap[] = {
```

```
/*  ?   Open       Read/Write   Close       Task #      Device  File
    -   ----       ----------   -----       -------     ------  ----      */
  DT(1, no_dev,    no_dev,      no_dev,     0)           /* 0 = not used   */
  DT(1, dev_opcl,  call_task,   dev_opcl,   MEM)         /* 1 = /dev/mem   */
  DT(1, dev_opcl,  call_task,   dev_opcl,   FLOPPY)      /* 2 = /dev/fd0   */
  DT(ENABLE_WINI,
        dev_opcl,  call_task,   dev_opcl,   WINCHESTER)  /* 3 = /dev/hd0   */
  DT(1, tty_open,  call_task,   dev_opcl,   TTY)         /* 4 = /dev/tty00 */
  DT(1, ctty_open, call_ctty,   ctty_close, TTY)         /* 5 = /dev/tty   */
  DT(1, dev_opcl,  call_task,   dev_opcl,    PRINTER)     /* 6 = /dev/lp    */

#if (MACHINE == IBM_PC)
  DT(ENABLE_NETWORKING,
        net_open,  call_task,   dev_opcl,   INET_PROC_NR)/* 7 = /dev/ip    */
  DT(ENABLE_CDROM,
        dev_opcl,  call_task,   dev_opcl,   CDROM)       /* 8 = /dev/cd0   */
  DT(0, 0,         0,           0,          0)           /* 9 = not used   */
  DT(ENABLE_SCSI,
        dev_opcl,  call_task,   dev_opcl,   SCSI)        /*10 = /dev/sd0   */
  DT(0, 0,         0,           0,          0)           /*11 = not used   */
  DT(0, 0,         0,           0,          0)           /*12 = not used   */
  DT(ENABLE_AUDIO,
        dev_opcl,  call_task,   dev_opcl,   AUDIO)       /*13 = /dev/audio */
  DT(ENABLE_AUDIO,
        dev_opcl,  call_task,   dev_opcl,   MIXER)       /*14 = /dev/mixer */
#endif /* IBM_PC */

#if (MACHINE == ATARI)
  DT(ENABLE_SCSI,
        dev_opcl,  call_task,   dev_opcl,   SCSI)        /* 7 = /dev/hdscsi0 */
#endif
};

PUBLIC int max_major = sizeof(dmap)/sizeof(struct dmap);
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/cache.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* The file system maintains a buffer cache to reduce the number of disk
 * accesses needed.  Whenever a read or write to the disk is done, a check is
 * first made to see if the block is in the cache.  This file manages the
 * cache.
 *
 * The entry points into this file are:
 *   get_block:   request to fetch a block for reading or writing from cache
 *   put_block:   return a block previously requested with get_block
 *   alloc_zone:  allocate a new zone (to increase the length of a file)
 *   free_zone:   release a zone (when a file is removed)
 *   rw_block:    read or write a block from the disk itself
 *   invalidate:  remove all the cache blocks on some device
 */

#include "fs.h"
#include <minix/com.h>
#include <minix/boot.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
```

```
#include "super.h"

FORWARD _PROTOTYPE( void rm_lru, (struct buf *bp) );

/*===========================================================================*
 *                              get_block                                    *
 *===========================================================================*/
PUBLIC struct buf *get_block(dev, block, only_search)
register dev_t dev;             /* on which device is the block? */
register block_t block;         /* which block is wanted? */
int only_search;                /* if NO_READ, don't read, else act normal */
{
/* Check to see if the requested block is in the block cache.  If so, return
 * a pointer to it.  If not, evict some other block and fetch it (unless
 * 'only_search' is 1).  All the blocks in the cache that are not in use
 * are linked together in a chain, with 'front' pointing to the least recently
 * used block and 'rear' to the most recently used block.  If 'only_search' is
 * 1, the block being requested will be overwritten in its entirety, so it is
 * only necessary to see if it is in the cache; if it is not, any free buffer
 * will do.  It is not necessary to actually read the block in from disk.
 * If 'only_search' is PREFETCH, the block need not be read from the disk,
 * and the device is not to be marked on the block, so callers can tell if
 * the block returned is valid.
 * In addition to the LRU chain, there is also a hash chain to link together
 * blocks whose block numbers end with the same bit strings, for fast lookup.
 */

  int b;
  register struct buf *bp, *prev_ptr;

  /* Search the hash chain for (dev, block). Do_read() can use
   * get_block(NO_DEV ...) to get an unnamed block to fill with zeros when
   * someone wants to read from a hole in a file, in which case this search
   * is skipped
   */
  if (dev != NO_DEV) {
        b = (int) block & HASH_MASK;
        bp = buf_hash[b];
        while (bp != NIL_BUF) {
                if (bp->b_blocknr == block && bp->b_dev == dev) {
                        /* Block needed has been found. */
                        if (bp->b_count == 0) rm_lru(bp);
                        bp->b_count++;  /* record that block is in use */
                        return(bp);
                } else {
                        /* This block is not the one sought. */
                        bp = bp->b_hash; /* move to next block on hash chain */
                }
        }
  }

  /* Desired block is not on available chain.  Take oldest block ('front'). */
  if ((bp = front) == NIL_BUF) panic("all buffers in use", NR_BUFS);
  rm_lru(bp);

  /* Remove the block that was just taken from its hash chain. */
  b = (int) bp->b_blocknr & HASH_MASK;
  prev_ptr = buf_hash[b];
  if (prev_ptr == bp) {
        buf_hash[b] = bp->b_hash;
```

```
  } else {
        /* The block just taken is not on the front of its hash chain. */
        while (prev_ptr->b_hash != NIL_BUF)
                if (prev_ptr->b_hash == bp) {
                        prev_ptr->b_hash = bp->b_hash;  /* found it */
                        break;
                } else {
                        prev_ptr = prev_ptr->b_hash;    /* keep looking */
                }
  }

  /* If the block taken is dirty, make it clean by writing it to the disk.
   * Avoid hysteresis by flushing all other dirty blocks for the same device.
   */
  if (bp->b_dev != NO_DEV) {
        if (bp->b_dirt == DIRTY) flushall(bp->b_dev);
  }

  /* Fill in block's parameters and add it to the hash chain where it goes. */
  bp->b_dev = dev;              /* fill in device number */
  bp->b_blocknr = block;        /* fill in block number */
  bp->b_count++;                /* record that block is being used */
  b = (int) bp->b_blocknr & HASH_MASK;
  bp->b_hash = buf_hash[b];
  buf_hash[b] = bp;             /* add to hash list */

  /* Go get the requested block unless searching or prefetching. */
  if (dev != NO_DEV) {
        if (only_search == PREFETCH) bp->b_dev = NO_DEV;
        else
        if (only_search == NORMAL) rw_block(bp, READING);
  }
  return(bp);                   /* return the newly acquired block */
}


/*===========================================================================*
 *                              put_block                                    *
 *===========================================================================*/
PUBLIC void put_block(bp, block_type)
register struct buf *bp;        /* pointer to the buffer to be released */
int block_type;                 /* INODE_BLOCK, DIRECTORY_BLOCK, or whatever */
{
/* Return a block to the list of available blocks.   Depending on 'block_type'
 * it may be put on the front or rear of the LRU chain.  Blocks that are
 * expected to be needed again shortly (e.g., partially full data blocks)
 * go on the rear; blocks that are unlikely to be needed again shortly
 * (e.g., full data blocks) go on the front.  Blocks whose loss can hurt
 * the integrity of the file system (e.g., inode blocks) are written to
 * disk immediately if they are dirty.
 */

  if (bp == NIL_BUF) return;    /* it is easier to check here than in caller */

  bp->b_count--;                /* there is one use fewer now */
  if (bp->b_count != 0) return; /* block is still in use */

  bufs_in_use--;                /* one fewer block buffers in use */

  /* Put this block back on the LRU chain.  If the ONE_SHOT bit is set in
```

```
   * 'block_type', the block is not likely to be needed again shortly, so put
   * it on the front of the LRU chain where it will be the first one to be
   * taken when a free buffer is needed later.
   */
  if (block_type & ONE_SHOT) {
        /* Block probably won't be needed quickly. Put it on front of chain.
         * It will be the next block to be evicted from the cache.
         */
        bp->b_prev = NIL_BUF;
        bp->b_next = front;
        if (front == NIL_BUF)
                rear = bp;      /* LRU chain was empty */
        else
                front->b_prev = bp;
        front = bp;
  } else {
        /* Block probably will be needed quickly.  Put it on rear of chain.
         * It will not be evicted from the cache for a long time.
         */
        bp->b_prev = rear;
        bp->b_next = NIL_BUF;
        if (rear == NIL_BUF)
                front = bp;
        else
                rear->b_next = bp;
        rear = bp;
  }

  /* Some blocks are so important (e.g., inodes, indirect blocks) that they
   * should be written to the disk immediately to avoid messing up the file
   * system in the event of a crash.
   */
  if ((block_type & WRITE_IMMED) && bp->b_dirt==DIRTY && bp->b_dev != NO_DEV)
        rw_block(bp, WRITING);
}


/*===========================================================================*
 *                              alloc_zone                                   *
 *===========================================================================*/
PUBLIC zone_t alloc_zone(dev, z)
dev_t dev;                      /* device where zone wanted */
zone_t z;                       /* try to allocate new zone near this one */
{
/* Allocate a new zone on the indicated device and return its number. */

  int major, minor;
  bit_t b, bit;
  struct super_block *sp;

  /* Note that the routine alloc_bit() returns 1 for the lowest possible
   * zone, which corresponds to sp->s_firstdatazone.  To convert a value
   * between the bit number, 'b', used by alloc_bit() and the zone number, 'z',
   * stored in the inode, use the formula:
   *     z = b + sp->s_firstdatazone - 1
   * Alloc_bit() never returns 0, since this is used for NO_BIT (failure).
   */
  sp = get_super(dev);          /* find the super_block for this device */

  /* If z is 0, skip initial part of the map known to be fully in use. */
```

```
  if (z == sp->s_firstdatazone) {
        bit = sp->s_zsearch;
  } else {
        bit = (bit_t) z - (sp->s_firstdatazone - 1);
  }
  b = alloc_bit(sp, ZMAP, bit);
  if (b == NO_BIT) {
        err_code = ENOSPC;
        major = (int) (sp->s_dev >> MAJOR) & BYTE;
        minor = (int) (sp->s_dev >> MINOR) & BYTE;
        printf("No space on %sdevice %d/%d\n",
                sp->s_dev == ROOT_DEV ? "root " : "", major, minor);
        return(NO_ZONE);
  }
  if (z == sp->s_firstdatazone) sp->s_zsearch = b;      /* for next time */
  return(sp->s_firstdatazone - 1 + (zone_t) b);
}


/*===========================================================================*
 *                              free_zone                                    *
 *===========================================================================*/
PUBLIC void free_zone(dev, numb)
dev_t dev;                              /* device where zone located */
zone_t numb;                            /* zone to be returned */
{
/* Return a zone. */

  register struct super_block *sp;
  bit_t bit;

  /* Locate the appropriate super_block and return bit. */
  sp = get_super(dev);
  if (numb < sp->s_firstdatazone || numb >= sp->s_zones) return;
  bit = (bit_t) (numb - (sp->s_firstdatazone - 1));
  free_bit(sp, ZMAP, bit);
  if (bit < sp->s_zsearch) sp->s_zsearch = bit;
}


/*===========================================================================*
 *                              rw_block                                     *
 *===========================================================================*/
PUBLIC void rw_block(bp, rw_flag)
register struct buf *bp;        /* buffer pointer */
int rw_flag;                    /* READING or WRITING */
{
/* Read or write a disk block. This is the only routine in which actual disk
 * I/O is invoked. If an error occurs, a message is printed here, but the error
 * is not reported to the caller.  If the error occurred while purging a block
 * from the cache, it is not clear what the caller could do about it anyway.
 */

  int r, op;
  off_t pos;
  dev_t dev;

  if ( (dev = bp->b_dev) != NO_DEV) {
        pos = (off_t) bp->b_blocknr * BLOCK_SIZE;
        op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
```

```
        r = dev_io(op, FALSE, dev, pos, BLOCK_SIZE, FS_PROC_NR, bp->b_data);
        if (r != BLOCK_SIZE) {
            if (r >= 0) r = END_OF_FILE;
            if (r != END_OF_FILE)
              printf("Unrecoverable disk error on device %d/%d, block %ld\n",
                        (dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
                bp->b_dev = NO_DEV;     /* invalidate block */

                /* Report read errors to interested parties. */
                if (rw_flag == READING) rdwt_err = r;
        }
  }

  bp->b_dirt = CLEAN;
}


/*===========================================================================*
 *                              invalidate                                   *
 *===========================================================================*/
PUBLIC void invalidate(device)
dev_t device;                   /* device whose blocks are to be purged */
{
/* Remove all the blocks belonging to some device from the cache. */

  register struct buf *bp;

  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
        if (bp->b_dev == device) bp->b_dev = NO_DEV;
}


/*===========================================================================*
 *                              flushall                                     *
 *===========================================================================*/
PUBLIC void flushall(dev)
dev_t dev;                      /* device to flush */
{
/* Flush all dirty blocks for one device. */

  register struct buf *bp;
  static struct buf *dirty[NR_BUFS];    /* static so it isn't on stack */
  int ndirty;

  for (bp = &buf[0], ndirty = 0; bp < &buf[NR_BUFS]; bp++)
        if (bp->b_dirt == DIRTY && bp->b_dev == dev) dirty[ndirty++] = bp;
  rw_scattered(dev, dirty, ndirty, WRITING);
}


/*===========================================================================*
 *                              rw_scattered                                 *
 *===========================================================================*/
PUBLIC void rw_scattered(dev, bufq, bufqsize, rw_flag)
dev_t dev;                      /* major-minor device number */
struct buf **bufq;              /* pointer to array of buffers */
int bufqsize;                   /* number of buffers */
int rw_flag;                    /* READING or WRITING */
{
/* Read or write scattered data from a device. */
```

```

  register struct buf *bp;
  int gap;
  register int i;
  register struct iorequest_s *iop;
  static struct iorequest_s iovec[NR_IOREQS];  /* static so it isn't on stack */
  int j;

  /* (Shell) sort buffers on b_blocknr. */
  gap = 1;
  do
	gap = 3 * gap + 1;
  while (gap <= bufqsize);
  while (gap != 1) {
	gap /= 3;
	for (j = gap; j < bufqsize; j++) {
		for (i = j - gap;
		     i >= 0 && bufq[i]->b_blocknr > bufq[i + gap]->b_blocknr;
		     i -= gap) {
			bp = bufq[i];
			bufq[i] = bufq[i + gap];
			bufq[i + gap] = bp;
		}
	}
  }

  /* Set up i/o vector and do i/o.  The result of dev_io is discarded because
   * all results are returned in the vector.  If dev_io fails completely, the
   * vector is unchanged and all results are taken as errors.
   */
  while (bufqsize > 0) {
	for (j = 0, iop = iovec; j < NR_IOREQS && j < bufqsize; j++, iop++) {
		bp = bufq[j];
		iop->io_position = (off_t) bp->b_blocknr * BLOCK_SIZE;
		iop->io_buf = bp->b_data;
		iop->io_nbytes = BLOCK_SIZE;
		iop->io_request = rw_flag == WRITING ?
				  DEV_WRITE : DEV_READ | OPTIONAL_IO;
	}
	(void) dev_io(SCATTERED_IO, 0, dev, (off_t) 0, j, FS_PROC_NR,
							(char *) iovec);

	/* Harvest the results.  Leave read errors for rw_block() to complain. */
	for (i = 0, iop = iovec; i < j; i++, iop++) {
		bp = bufq[i];
		if (rw_flag == READING) {
		    if (iop->io_nbytes == 0)
			bp->b_dev = dev;	/* validate block */
		    put_block(bp, PARTIAL_DATA_BLOCK);
		} else {
		    if (iop->io_nbytes != 0) {
		     printf("Unrecoverable write error on device %d/%d, block %ld\n",
				(dev>>MAJOR)&BYTE, (dev>>MINOR)&BYTE, bp->b_blocknr);
			bp->b_dev = NO_DEV;	/* invalidate block */
		    }
		    bp->b_dirt = CLEAN;
		}
	}
	bufq += j;
	bufqsize -= j;
```

```
  }
}


/*===========================================================================*
 *                              rm_lru                                       *
 *===========================================================================*/
PRIVATE void rm_lru(bp)
struct buf *bp;
{
/* Remove a block from its LRU chain. */

  struct buf *next_ptr, *prev_ptr;

  bufs_in_use++;
  next_ptr = bp->b_next;        /* successor on LRU chain */
  prev_ptr = bp->b_prev;        /* predecessor on LRU chain */
  if (prev_ptr != NIL_BUF)
        prev_ptr->b_next = next_ptr;
  else
        front = next_ptr;       /* this block was at front of chain */

  if (next_ptr != NIL_BUF)
        next_ptr->b_prev = prev_ptr;
  else
        rear = prev_ptr;        /* this block was at rear of chain */
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/inode.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file manages the inode table.  There are procedures to allocate and
 * deallocate inodes, acquire, erase, and release them, and read and write
 * them from the disk.
 *
 * The entry points into this file are
 *   get_inode:    search inode table for a given inode; if not there, read it
 *   put_inode:    indicate that an inode is no longer needed in memory
 *   alloc_inode:  allocate a new, unused inode
 *   wipe_inode:   erase some fields of a newly allocated inode
 *   free_inode:   mark an inode as available for a new file
 *   update_times: update atime, ctime, and mtime
 *   rw_inode:     read a disk block and extract an inode, or corresp. write
 *   old_icopy:    copy to/from in-core inode struct and disk inode (V1.x)
 *   new_icopy:    copy to/from in-core inode struct and disk inode (V2.x)
 *   dup_inode:    indicate that someone else is using an inode table entry
 */

#include "fs.h"
#include <minix/boot.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

```

```
FORWARD _PROTOTYPE( void old_icopy, (struct inode *rip, d1_inode *dip,
                                                int direction, int norm));
FORWARD _PROTOTYPE( void new_icopy, (struct inode *rip, d2_inode *dip,
                                                int direction, int norm));


/*===========================================================================*
 *                              get_inode                                    *
 *===========================================================================*/
PUBLIC struct inode *get_inode(dev, numb)
dev_t dev;                      /* device on which inode resides */
int numb;                       /* inode number (ANSI: may not be unshort) */
{
/* Find a slot in the inode table, load the specified inode into it, and
 * return a pointer to the slot.  If 'dev' == NO_DEV, just return a free slot.
 */

  register struct inode *rip, *xp;

  /* Search the inode table both for (dev, numb) and a free slot. */
  xp = NIL_INODE;
  for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++) {
        if (rip->i_count > 0) { /* only check used slots for (dev, numb) */
                if (rip->i_dev == dev && rip->i_num == numb) {
                        /* This is the inode that we are looking for. */
                        rip->i_count++;
                        return(rip);    /* (dev, numb) found */
                }
        } else {
                xp = rip;       /* remember this free slot for later */
        }
  }

  /* Inode we want is not currently in use.  Did we find a free slot? */
  if (xp == NIL_INODE) {        /* inode table completely full */
        err_code = ENFILE;
        return(NIL_INODE);
  }

  /* A free inode slot has been located.  Load the inode into it. */
  xp->i_dev = dev;
  xp->i_num = numb;
  xp->i_count = 1;
  if (dev != NO_DEV) rw_inode(xp, READING);     /* get inode from disk */
  xp->i_update = 0;             /* all the times are initially up-to-date */

  return(xp);
}


/*===========================================================================*
 *                              put_inode                                    *
 *===========================================================================*/
PUBLIC void put_inode(rip)
register struct inode *rip;     /* pointer to inode to be released */
{
/* The caller is no longer using this inode.  If no one else is using it either
 * write it back to the disk immediately.  If it has no links, truncate it and
 * return it to the pool of available inodes.
 */
```

```

  if (rip == NIL_INODE) return; /* checking here is easier than in caller */
  if (--rip->i_count == 0) {    /* i_count == 0 means no one is using it now */
        if ((rip->i_nlinks & BYTE) == 0) {
                /* i_nlinks == 0 means free the inode. */
                truncate(rip);  /* return all the disk blocks */
                rip->i_mode = I_NOT_ALLOC;      /* clear I_TYPE field */
                rip->i_dirt = DIRTY;
                free_inode(rip->i_dev, rip->i_num);
        } else {
                if (rip->i_pipe == I_PIPE) truncate(rip);
        }
        rip->i_pipe = NO_PIPE;  /* should always be cleared */
        if (rip->i_dirt == DIRTY) rw_inode(rip, WRITING);
  }
}

/*===========================================================================*
 *                              alloc_inode                                  *
 *===========================================================================*/
PUBLIC struct inode *alloc_inode(dev, bits)
dev_t dev;                      /* device on which to allocate the inode */
mode_t bits;                    /* mode of the inode */
{
/* Allocate a free inode on 'dev', and return a pointer to it. */

  register struct inode *rip;
  register struct super_block *sp;
  int major, minor, inumb;
  bit_t b;

  sp = get_super(dev);  /* get pointer to super_block */
  if (sp->s_rd_only) {  /* can't allocate an inode on a read only device. */
        err_code = EROFS;
        return(NIL_INODE);
  }

  /* Acquire an inode from the bit map. */
  b = alloc_bit(sp, IMAP, sp->s_isearch);
  if (b == NO_BIT) {
        err_code = ENFILE;
        major = (int) (sp->s_dev >> MAJOR) & BYTE;
        minor = (int) (sp->s_dev >> MINOR) & BYTE;
        printf("Out of i-nodes on %sdevice %d/%d\n",
                sp->s_dev == ROOT_DEV ? "root " : "", major, minor);
        return(NIL_INODE);
  }
  sp->s_isearch = b;            /* next time start here */
  inumb = (int) b;              /* be careful not to pass unshort as param */

  /* Try to acquire a slot in the inode table. */
  if ((rip = get_inode(NO_DEV, inumb)) == NIL_INODE) {
        /* No inode table slots available.  Free the inode just allocated. */
        free_bit(sp, IMAP, b);
  } else {
        /* An inode slot is available. Put the inode just allocated into it. */
        rip->i_mode = bits;             /* set up RWX bits */
        rip->i_nlinks = (nlink_t) 0;    /* initial no links */
        rip->i_uid = fp->fp_effuid;     /* file's uid is owner's */
        rip->i_gid = fp->fp_effgid;     /* ditto group id */
```

```
        rip->i_dev = dev;               /* mark which device it is on */
        rip->i_ndzones = sp->s_ndzones; /* number of direct zones */
        rip->i_nindirs = sp->s_nindirs; /* number of indirect zones per blk*/
        rip->i_sp = sp;                 /* pointer to super block */

        /* Fields not cleared already are cleared in wipe_inode().  They have
         * been put there because truncate() needs to clear the same fields if
         * the file happens to be open while being truncated.  It saves space
         * not to repeat the code twice.
         */
        wipe_inode(rip);
  }

  return(rip);
}

/*===========================================================================*
 *                              wipe_inode                                   *
 *===========================================================================*/
PUBLIC void wipe_inode(rip)
register struct inode *rip;     /* the inode to be erased */
{
/* Erase some fields in the inode.  This function is called from alloc_inode()
 * when a new inode is to be allocated, and from truncate(), when an existing
 * inode is to be truncated.
 */

  register int i;

  rip->i_size = 0;
  rip->i_update = ATIME | CTIME | MTIME;        /* update all times later */
  rip->i_dirt = DIRTY;
  for (i = 0; i < V2_NR_TZONES; i++) rip->i_zone[i] = NO_ZONE;
}


/*===========================================================================*
 *                              free_inode                                   *
 *===========================================================================*/
PUBLIC void free_inode(dev, inumb)
dev_t dev;                      /* on which device is the inode */
ino_t inumb;                    /* number of inode to be freed */
{
/* Return an inode to the pool of unallocated inodes. */

  register struct super_block *sp;
  bit_t b;

  /* Locate the appropriate super_block. */
  sp = get_super(dev);
  if (inumb <= 0 || inumb > sp->s_ninodes) return;
  b = inumb;
  free_bit(sp, IMAP, b);
  if (b < sp->s_isearch) sp->s_isearch = b;
}

/*===========================================================================*
 *                              update_times                                 *
 *===========================================================================*/
PUBLIC void update_times(rip)
```

```
register struct inode *rip;     /* pointer to inode to be read/written */
{
/* Various system calls are required by the standard to update atime, ctime,
 * or mtime.  Since updating a time requires sending a message to the clock
 * task--an expensive business--the times are marked for update by setting
 * bits in i_update.  When a stat, fstat, or sync is done, or an inode is
 * released, update_times() may be called to actually fill in the times.
 */

  time_t cur_time;
  struct super_block *sp;

  sp = rip->i_sp;               /* get pointer to super block. */
  if (sp->s_rd_only) return;    /* no updates for read-only file systems */

  cur_time = clock_time();
  if (rip->i_update & ATIME) rip->i_atime = cur_time;
  if (rip->i_update & CTIME) rip->i_ctime = cur_time;
  if (rip->i_update & MTIME) rip->i_mtime = cur_time;
  rip->i_update = 0;            /* they are all up-to-date now */
}


/*===========================================================================*
 *                              rw_inode                                     *
 *===========================================================================*/
PUBLIC void rw_inode(rip, rw_flag)
register struct inode *rip;     /* pointer to inode to be read/written */
int rw_flag;                    /* READING or WRITING */
{
/* An entry in the inode table is to be copied to or from the disk. */

  register struct buf *bp;
  register struct super_block *sp;
  d1_inode *dip;
  d2_inode *dip2;
  block_t b, offset;

  /* Get the block where the inode resides. */
  sp = get_super(rip->i_dev);   /* get pointer to super block */
  rip->i_sp = sp;               /* inode must contain super block pointer */
  offset = sp->s_imap_blocks + sp->s_zmap_blocks + 2;
  b = (block_t) (rip->i_num - 1)/sp->s_inodes_per_block + offset;
  bp = get_block(rip->i_dev, b, NORMAL);
  dip  = bp->b_v1_ino + (rip->i_num - 1) % V1_INODES_PER_BLOCK;
  dip2 = bp->b_v2_ino + (rip->i_num - 1) % V2_INODES_PER_BLOCK;

  /* Do the read or write. */
  if (rw_flag == WRITING) {
        if (rip->i_update) update_times(rip);   /* times need updating */
        if (sp->s_rd_only == FALSE) bp->b_dirt = DIRTY;
  }

  /* Copy the inode from the disk block to the in-core table or vice versa.
   * If the fourth parameter below is FALSE, the bytes are swapped.
   */
  if (sp->s_version == V1)
        old_icopy(rip, dip,  rw_flag, sp->s_native);
  else
        new_icopy(rip, dip2, rw_flag, sp->s_native);
```

```
  put_block(bp, INODE_BLOCK);
  rip->i_dirt = CLEAN;
}


/*===========================================================================*
 *                               old_icopy                                   *
 *===========================================================================*/
PRIVATE void old_icopy(rip, dip, direction, norm)
register struct inode *rip;     /* pointer to the in-core inode struct */
register d1_inode *dip;         /* pointer to the d1_inode inode struct */
int direction;                  /* READING (from disk) or WRITING (to disk) */
int norm;                       /* TRUE = do not swap bytes; FALSE = swap */

{
/* The V1.x IBM disk, the V1.x 68000 disk, and the V2 disk (same for IBM and
 * 68000) all have different inode layouts.  When an inode is read or written
 * this routine handles the conversions so that the information in the inode
 * table is independent of the disk structure from which the inode came.
 * The old_icopy routine copies to and from V1 disks.
 */

  int i;

  if (direction == READING) {
        /* Copy V1.x inode to the in-core table, swapping bytes if need be. */
        rip->i_mode    = conv2(norm, (int) dip->d1_mode);
        rip->i_uid     = conv2(norm, (int) dip->d1_uid );
        rip->i_size    = conv4(norm,       dip->d1_size);
        rip->i_mtime   = conv4(norm,       dip->d1_mtime);
        rip->i_atime   = rip->i_mtime;
        rip->i_ctime   = rip->i_mtime;
        rip->i_nlinks  = (nlink_t) dip->d1_nlinks;      /* 1 char */
        rip->i_gid     = (gid_t) dip->d1_gid;           /* 1 char */
        rip->i_ndzones = V1_NR_DZONES;
        rip->i_nindirs = V1_INDIRECTS;
        for (i = 0; i < V1_NR_TZONES; i++)
                rip->i_zone[i] = conv2(norm, (int) dip->d1_zone[i]);
  } else {
        /* Copying V1.x inode to disk from the in-core table. */
        dip->d1_mode   = conv2(norm, (int) rip->i_mode);
        dip->d1_uid    = conv2(norm, (int) rip->i_uid );
        dip->d1_size   = conv4(norm,       rip->i_size);
        dip->d1_mtime  = conv4(norm,       rip->i_mtime);
        dip->d1_nlinks = (nlink_t) rip->i_nlinks;       /* 1 char */
        dip->d1_gid    = (gid_t) rip->i_gid;            /* 1 char */
        for (i = 0; i < V1_NR_TZONES; i++)
                dip->d1_zone[i] = conv2(norm, (int) rip->i_zone[i]);
  }
}


/*===========================================================================*
 *                               new_icopy                                   *
 *===========================================================================*/
PRIVATE void new_icopy(rip, dip, direction, norm)
register struct inode *rip;     /* pointer to the in-core inode struct */
register d2_inode *dip; /* pointer to the d2_inode struct */
int direction;                  /* READING (from disk) or WRITING (to disk) */
```

```
int norm;                       /* TRUE = do not swap bytes; FALSE = swap */

{
/* Same as old_icopy, but to/from V2 disk layout. */

  int i;

  if (direction == READING) {
        /* Copy V2.x inode to the in-core table, swapping bytes if need be. */
        rip->i_mode    = conv2(norm,dip->d2_mode);
        rip->i_uid     = conv2(norm,dip->d2_uid );
        rip->i_nlinks  = conv2(norm,(int) dip->d2_nlinks);
        rip->i_gid     = conv2(norm,(int) dip->d2_gid );
        rip->i_size    = conv4(norm,dip->d2_size);
        rip->i_atime   = conv4(norm,dip->d2_atime);
        rip->i_ctime   = conv4(norm,dip->d2_ctime);
        rip->i_mtime   = conv4(norm,dip->d2_mtime);
        rip->i_ndzones = V2_NR_DZONES;
        rip->i_nindirs = V2_INDIRECTS;
        for (i = 0; i < V2_NR_TZONES; i++)
                rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
  } else {
        /* Copying V2.x inode to disk from the in-core table. */
        dip->d2_mode   = conv2(norm,rip->i_mode);
        dip->d2_uid    = conv2(norm,rip->i_uid );
        dip->d2_nlinks = conv2(norm,rip->i_nlinks);
        dip->d2_gid    = conv2(norm,rip->i_gid );
        dip->d2_size   = conv4(norm,rip->i_size);
        dip->d2_atime  = conv4(norm,rip->i_atime);
        dip->d2_ctime  = conv4(norm,rip->i_ctime);
        dip->d2_mtime  = conv4(norm,rip->i_mtime);
        for (i = 0; i < V2_NR_TZONES; i++)
                dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
  }
}


/*===========================================================================*
 *                              dup_inode                                    *
 *===========================================================================*/
PUBLIC void dup_inode(ip)
struct inode *ip;               /* The inode to be duplicated. */
{
/* This routine is a simplified form of get_inode() for the case where
 * the inode pointer is already known.
 */

  ip->i_count++;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/super.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file manages the super block table and the related data structures,
 * namely, the bit maps that keep track of which zones and which inodes are
 * allocated and which are free.  When a new inode or zone is needed, the
 * appropriate bit map is searched for a free entry.
 *
 * The entry points into this file are
 *   alloc_bit:       somebody wants to allocate a zone or inode; find one
 *   free_bit:        indicate that a zone or inode is available for allocation
 *   get_super:       search the 'superblock' table for a device
 *   mounted:         tells if file inode is on mounted (or ROOT) file system
 *   read_super:      read a superblock
 */

#include "fs.h"
#include <string.h>
#include <minix/boot.h>
#include "buf.h"
#include "inode.h"
#include "super.h"

#define BITCHUNK_BITS   (usizeof(bitchunk_t) * CHAR_BIT)
#define BITS_PER_BLOCK  (BITMAP_CHUNKS * BITCHUNK_BITS)

/*===========================================================================*
 *                              alloc_bit                                    *
 *===========================================================================*/
PUBLIC bit_t alloc_bit(sp, map, origin)
struct super_block *sp;         /* the filesystem to allocate from */
int map;                        /* IMAP (inode map) or ZMAP (zone map) */
bit_t origin;                   /* number of bit to start searching at */
{
/* Allocate a bit from a bit map and return its bit number. */

  block_t start_block;          /* first bit block */
  bit_t map_bits;               /* how many bits are there in the bit map? */
  unsigned bit_blocks;          /* how many blocks are there in the bit map? */
  unsigned block, word, bcount, wcount;
  struct buf *bp;
  bitchunk_t *wptr, *wlim, k;
  bit_t i, b;

  if (sp->s_rd_only)
        panic("can't allocate bit on read-only filesys.", NO_NUM);

  if (map == IMAP) {
        start_block = SUPER_BLOCK + 1;
        map_bits = sp->s_ninodes + 1;
        bit_blocks = sp->s_imap_blocks;
  } else {
        start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
        map_bits = sp->s_zones - (sp->s_firstdatazone - 1);
        bit_blocks = sp->s_zmap_blocks;
  }

  /* Figure out where to start the bit search (depends on 'origin'). */
```

```
  if (origin >= map_bits) origin = 0;   /* for robustness */

  /* Locate the starting place. */
  block = origin / BITS_PER_BLOCK;
  word = (origin % BITS_PER_BLOCK) / BITCHUNK_BITS;

  /* Iterate over all blocks plus one, because we start in the middle. */
  bcount = bit_blocks + 1;
  do {
        bp = get_block(sp->s_dev, start_block + block, NORMAL);
        wlim = &bp->b_bitmap[BITMAP_CHUNKS];

        /* Iterate over the words in block. */
        for (wptr = &bp->b_bitmap[word]; wptr < wlim; wptr++) {

                /* Does this word contain a free bit? */
                if (*wptr == (bitchunk_t) ˜0) continue;

                /* Find and allocate the free bit. */
                k = conv2(sp->s_native, (int) *wptr);
                for (i = 0; (k & (1 << i)) != 0; ++i) {}

                /* Bit number from the start of the bit map. */
                b = ((bit_t) block * BITS_PER_BLOCK)
                    + (wptr - &bp->b_bitmap[0]) * BITCHUNK_BITS
                    + i;

                /* Don't allocate bits beyond the end of the map. */
                if (b >= map_bits) break;

                /* Allocate and return bit number. */
                k |= 1 << i;
                *wptr = conv2(sp->s_native, (int) k);
                bp->b_dirt = DIRTY;
                put_block(bp, MAP_BLOCK);
                return(b);
        }
        put_block(bp, MAP_BLOCK);
        if (++block >= bit_blocks) block = 0;   /* last block, wrap around */
        word = 0;
  } while (--bcount > 0);
  return(NO_BIT);               /* no bit could be allocated */
}


/*===========================================================================*
 *                              free_bit                                     *
 *===========================================================================*/
PUBLIC void free_bit(sp, map, bit_returned)
struct super_block *sp;         /* the filesystem to operate on */
int map;                        /* IMAP (inode map) or ZMAP (zone map) */
bit_t bit_returned;             /* number of bit to insert into the map */
{
/* Return a zone or inode by turning off its bitmap bit. */

  unsigned block, word, bit;
  struct buf *bp;
  bitchunk_t k, mask;
  block_t start_block;

```

```
  if (sp->s_rd_only)
        panic("can’t free bit on read-only filesys.", NO_NUM);

  if (map == IMAP) {
        start_block = SUPER_BLOCK + 1;
  } else {
        start_block = SUPER_BLOCK + 1 + sp->s_imap_blocks;
  }
  block = bit_returned / BITS_PER_BLOCK;
  word = (bit_returned % BITS_PER_BLOCK) / BITCHUNK_BITS;
  bit = bit_returned % BITCHUNK_BITS;
  mask = 1 << bit;

  bp = get_block(sp->s_dev, start_block + block, NORMAL);

  k = conv2(sp->s_native, (int) bp->b_bitmap[word]);
  if (!(k & mask)) {
        panic(map == IMAP ? "tried to free unused inode" :
              "tried to free unused block", NO_NUM);
  }

  k &= ˜mask;
  bp->b_bitmap[word] = conv2(sp->s_native, (int) k);
  bp->b_dirt = DIRTY;

  put_block(bp, MAP_BLOCK);
}


/*===========================================================================*
 *                              get_super                                    *
 *===========================================================================*/
PUBLIC struct super_block *get_super(dev)
dev_t dev;                      /* device number whose super_block is sought */
{
/* Search the superblock table for this device.  It is supposed to be there. */

  register struct super_block *sp;

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
        if (sp->s_dev == dev) return(sp);

  /* Search failed.  Something wrong. */
  panic("can’t find superblock for device (in decimal)", (int) dev);

  return(NIL_SUPER);            /* to keep the compiler and lint quiet */
}


/*===========================================================================*
 *                              mounted                                      *
 *===========================================================================*/
PUBLIC int mounted(rip)
register struct inode *rip;     /* pointer to inode */
{
/* Report on whether the given inode is on a mounted (or ROOT) file system. */

  register struct super_block *sp;
  register dev_t dev;

```

```
  dev = (dev_t) rip->i_zone[0];
  if (dev == ROOT_DEV) return(TRUE);    /* inode is on root file system */

  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
        if (sp->s_dev == dev) return(TRUE);

  return(FALSE);
}


/*===========================================================================*
 *                              read_super                                   *
 *===========================================================================*/
PUBLIC int read_super(sp)
register struct super_block *sp; /* pointer to a superblock */
{
/* Read a superblock. */

  register struct buf *bp;
  dev_t dev;
  int magic;
  int version, native;

  dev = sp->s_dev;              /* save device (will be overwritten by copy) */
  bp = get_block(sp->s_dev, SUPER_BLOCK, NORMAL);
  memcpy( (char *) sp, bp->b_data, (size_t) SUPER_SIZE);
  put_block(bp, ZUPER_BLOCK);
  sp->s_dev = NO_DEV;           /* restore later */
  magic = sp->s_magic;          /* determines file system type */

  /* Get file system version and type. */
  if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
        version = V1;
        native  = (magic == SUPER_MAGIC);
  } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
        version = V2;
        native  = (magic == SUPER_V2);
  } else {
        return(EINVAL);
  }

  /* If the super block has the wrong byte order, swap the fields; the magic
   * number doesn't need conversion. */
  sp->s_ninodes =       conv2(native, (int) sp->s_ninodes);
  sp->s_nzones =        conv2(native, (int) sp->s_nzones);
  sp->s_imap_blocks =   conv2(native, (int) sp->s_imap_blocks);
  sp->s_zmap_blocks =   conv2(native, (int) sp->s_zmap_blocks);
  sp->s_firstdatazone = conv2(native, (int) sp->s_firstdatazone);
  sp->s_log_zone_size = conv2(native, (int) sp->s_log_zone_size);
  sp->s_max_size =      conv4(native, sp->s_max_size);
  sp->s_zones =         conv4(native, sp->s_zones);

  /* In V1, the device size was kept in a short, s_nzones, which limited
   * devices to 32K zones.  For V2, it was decided to keep the size as a
   * long.  However, just changing s_nzones to a long would not work, since
   * then the position of s_magic in the super block would not be the same
   * in V1 and V2 file systems, and there would be no way to tell whether
   * a newly mounted file system was V1 or V2.  The solution was to introduce
   * a new variable, s_zones, and copy the size there.
   *
```

```
   * Calculate some other numbers that depend on the version here too, to
   * hide some of the differences.
   */
  if (version == V1) {
        sp->s_zones = sp->s_nzones;     /* only V1 needs this copy */
        sp->s_inodes_per_block = V1_INODES_PER_BLOCK;
        sp->s_ndzones = V1_NR_DZONES;
        sp->s_nindirs = V1_INDIRECTS;
  } else {
        sp->s_inodes_per_block = V2_INODES_PER_BLOCK;
        sp->s_ndzones = V2_NR_DZONES;
        sp->s_nindirs = V2_INDIRECTS;
  }

  sp->s_isearch = 0;            /* inode searches initially start at 0 */
  sp->s_zsearch = 0;            /* zone searches initially start at 0 */
  sp->s_version = version;
  sp->s_native  = native;

  /* Make a few basic checks to see if super block looks reasonable. */
  if (sp->s_imap_blocks < 1 || sp->s_zmap_blocks < 1
                                || sp->s_ninodes < 1 || sp->s_zones < 1
                                || (unsigned) sp->s_log_zone_size > 4) {
        return(EINVAL);
  }
  sp->s_dev = dev;              /* restore device number */
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  src/fs/filedes.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the procedures that manipulate file descriptors.
 *
 * The entry points into this file are
 *   get_fd:    look for free file descriptor and free filp slots
 *   get_filp:  look up the filp entry for a given file descriptor
 *   find_filp: find a filp slot that points to a given inode
 */

#include "fs.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"

/*===========================================================================*
 *                              get_fd                                       *
 *===========================================================================*/
PUBLIC int get_fd(start, bits, k, fpt)
int start;                      /* start of search (used for F_DUPFD) */
mode_t bits;                    /* mode of the file to be created (RWX bits) */
int *k;                         /* place to return file descriptor */
struct filp **fpt;              /* place to return filp slot */
{
/* Look for a free file descriptor and a free filp slot.  Fill in the mode word
 * in the latter, but don't claim either one yet, since the open() or creat()
 * may yet fail.
```

```
 */

  register struct filp *f;
  register int i;

  *k = -1;                      /* we need a way to tell if file desc found */

  /* Search the fproc fp_filp table for a free file descriptor. */
  for (i = start; i < OPEN_MAX; i++) {
        if (fp->fp_filp[i] == NIL_FILP) {
                /* A file descriptor has been located. */
                *k = i;
                break;
        }
  }

  /* Check to see if a file descriptor has been found. */
  if (*k < 0) return(EMFILE);   /* this is why we initialized k to -1 */

  /* Now that a file descriptor has been found, look for a free filp slot. */
  for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
        if (f->filp_count == 0) {
                f->filp_mode = bits;
                f->filp_pos = 0L;
                f->filp_flags = 0;
                *fpt = f;
                return(OK);
        }
  }

  /* If control passes here, the filp table must be full.  Report that back. */
  return(ENFILE);
}


/*===========================================================================*
 *                              get_filp                                     *
 *===========================================================================*/
PUBLIC struct filp *get_filp(fild)
int fild;                       /* file descriptor */
{
/* See if 'fild' refers to a valid file descr.  If so, return its filp ptr. */

  err_code = EBADF;
  if (fild < 0 || fild >= OPEN_MAX ) return(NIL_FILP);
  return(fp->fp_filp[fild]);    /* may also be NIL_FILP */
}


/*===========================================================================*
 *                              find_filp                                    *
 *===========================================================================*/
PUBLIC struct filp *find_filp(rip, bits)
register struct inode *rip;     /* inode referred to by the filp to be found */
Mode_t bits;                    /* mode of the filp to be found (RWX bits) */
{
/* Find a filp slot that refers to the inode 'rip' in a way as described
 * by the mode bit 'bits'. Used for determining whether somebody is still
 * interested in either end of a pipe.  Also used when opening a FIFO to
 * find partners to share a filp field with (to shared the file position).
```

```
 * Like 'get_fd' it performs its job by linear search through the filp table.
 */

  register struct filp *f;

  for (f = &filp[0]; f < &filp[NR_FILPS]; f++) {
        if (f->filp_count != 0 && f->filp_ino == rip && (f->filp_mode & bits)){
                return(f);
        }
  }

  /* If control passes here, the filp wasn't there.  Report that back. */
  return(NIL_FILP);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/lock.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file handles advisory file locking as required by POSIX.
 *
 * The entry points into this file are
 *   lock_op:   perform locking operations for FCNTL system call
 *   lock_revive: revive processes when a lock is released
 */

#include "fs.h"
#include <fcntl.h>
#include <unistd.h>     /* cc runs out of memory with unistd.h :-( */
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "param.h"

/*===========================================================================*
 *                              lock_op                                      *
 *===========================================================================*/
PUBLIC int lock_op(f, req)
struct filp *f;
int req;                        /* either F_SETLK or F_SETLKW */
{
/* Perform the advisory locking required by POSIX. */

  int r, ltype, i, conflict = 0, unlocking = 0;
  mode_t mo;
  off_t first, last;
  struct flock flock;
  vir_bytes user_flock;
  struct file_lock *flp, *flp2, *empty;

  /* Fetch the flock structure from user space. */
  user_flock = (vir_bytes) name1;
  r = sys_copy(who, D, (phys_bytes) user_flock,
        FS_PROC_NR, D, (phys_bytes) &flock, (phys_bytes) sizeof(flock));
  if (r != OK) return(EINVAL);

  /* Make some error checks. */
  ltype = flock.l_type;
```

```
        mo = f->filp_mode;
        if (ltype != F_UNLCK && ltype != F_RDLCK && ltype != F_WRLCK) return(EINVAL);
        if (req == F_GETLK && ltype == F_UNLCK) return(EINVAL);
        if ( (f->filp_ino->i_mode & I_TYPE) != I_REGULAR) return(EINVAL);
        if (req != F_GETLK && ltype == F_RDLCK && (mo & R_BIT) == 0) return(EBADF);
        if (req != F_GETLK && ltype == F_WRLCK && (mo & W_BIT) == 0) return(EBADF);

        /* Compute the first and last bytes in the lock region. */
        switch (flock.l_whence) {
                case SEEK_SET:  first = 0; break;
                case SEEK_CUR:  first = f->filp_pos; break;
                case SEEK_END:  first = f->filp_ino->i_size; break;
                default:        return(EINVAL);
        }
        /* Check for overflow. */
        if (((long)flock.l_start > 0) && ((first + flock.l_start) < first))
                return(EINVAL);
        if (((long)flock.l_start < 0) && ((first + flock.l_start) > first))
                return(EINVAL);
        first = first + flock.l_start;
        last = first + flock.l_len - 1;
        if (flock.l_len == 0) last = MAX_FILE_POS;
        if (last < first) return(EINVAL);

        /* Check if this region conflicts with any existing lock. */
        empty = (struct file_lock *) 0;
        for (flp = &file_lock[0]; flp < & file_lock[NR_LOCKS]; flp++) {
                if (flp->lock_type == 0) {
                        if (empty == (struct file_lock *) 0) empty = flp;
                        continue;       /* 0 means unused slot */
                }
                if (flp->lock_inode != f->filp_ino) continue;   /* different file */
                if (last < flp->lock_first) continue;   /* new one is in front */
                if (first > flp->lock_last) continue;   /* new one is afterwards */
                if (ltype == F_RDLCK && flp->lock_type == F_RDLCK) continue;
                if (ltype != F_UNLCK && flp->lock_pid == fp->fp_pid) continue;

                /* There might be a conflict.  Process it. */
                conflict = 1;
                if (req == F_GETLK) break;

                /* If we are trying to set a lock, it just failed. */
                if (ltype == F_RDLCK || ltype == F_WRLCK) {
                        if (req == F_SETLK) {
                                /* For F_SETLK, just report back failure. */
                                return(EAGAIN);
                        } else {
                                /* For F_SETLKW, suspend the process. */
                                suspend(XLOCK);
                                return(0);
                        }
                }

                /* We are clearing a lock and we found something that overlaps. */
                unlocking = 1;
                if (first <= flp->lock_first && last >= flp->lock_last) {
                        flp->lock_type = 0;     /* mark slot as unused */
                        nr_locks--;             /* number of locks is now 1 less */
                        continue;
                }
```

```

        /* Part of a locked region has been unlocked. */
        if (first <= flp->lock_first) {
                flp->lock_first = last + 1;
                continue;
        }

        if (last >= flp->lock_last) {
                flp->lock_last = first - 1;
                continue;
        }

        /* Bad luck. A lock has been split in two by unlocking the middle. */
        if (nr_locks == NR_LOCKS) return(ENOLCK);
        for (i = 0; i < NR_LOCKS; i++)
                if (file_lock[i].lock_type == 0) break;
        flp2 = &file_lock[i];
        flp2->lock_type = flp->lock_type;
        flp2->lock_pid = flp->lock_pid;
        flp2->lock_inode = flp->lock_inode;
        flp2->lock_first = last + 1;
        flp2->lock_last = flp->lock_last;
        flp->lock_last = first - 1;
        nr_locks++;
  }
  if (unlocking) lock_revive();

  if (req == F_GETLK) {
        if (conflict) {
                /* GETLK and conflict. Report on the conflicting lock. */
                flock.l_type = flp->lock_type;
                flock.l_whence = SEEK_SET;
                flock.l_start = flp->lock_first;
                flock.l_len = flp->lock_last - flp->lock_first + 1;
                flock.l_pid = flp->lock_pid;

        } else {
                /* It is GETLK and there is no conflict. */
                flock.l_type = F_UNLCK;
        }

        /* Copy the flock structure back to the caller. */
        r = sys_copy(FS_PROC_NR, D, (phys_bytes) &flock,
                who, D, (phys_bytes) user_flock, (phys_bytes) sizeof(flock));
        return(r);
  }

  if (ltype == F_UNLCK) return(OK);     /* unlocked a region with no locks */

  /* There is no conflict.  If space exists, store new lock in the table. */
  if (empty == (struct file_lock *) 0) return(ENOLCK);  /* table full */
  empty->lock_type = ltype;
  empty->lock_pid = fp->fp_pid;
  empty->lock_inode = f->filp_ino;
  empty->lock_first = first;
  empty->lock_last = last;
  nr_locks++;
  return(OK);
}
```

```
/*===========================================================================*
 *                              lock_revive                                  *
 *===========================================================================*/
PUBLIC void lock_revive()
{
/* Go find all the processes that are waiting for any kind of lock and
 * revive them all.  The ones that are still blocked will block again when
 * they run.  The others will complete.  This strategy is a space-time
 * tradeoff.  Figuring out exactly which ones to unblock now would take
 * extra code, and the only thing it would win would be some performance in
 * extremely rare circumstances (namely, that somebody actually used
 * locking).
 */

  int task;
  struct fproc *fptr;

  for (fptr = &fproc[INIT_PROC_NR + 1]; fptr < &fproc[NR_PROCS]; fptr++){
        task = -fptr->fp_task;
        if (fptr->fp_suspended == SUSPENDED && task == XLOCK) {
                revive( (int) (fptr - fproc), 0);
        }
  }
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/main.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the main program of the File System.  It consists of
 * a loop that gets messages requesting work, carries out the work, and sends
 * replies.
 *
 * The entry points into this file are
 *   main:      main program of the File System
 *   reply:     send a reply to a process after the requested work is done
 */

struct super_block;             /* proto.h needs to know this */

#include "fs.h"
#include <fcntl.h>
#include <string.h>
#include <sys/ioctl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/boot.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

FORWARD _PROTOTYPE( void buf_pool, (void)                               );
FORWARD _PROTOTYPE( void fs_init, (void)                                );
FORWARD _PROTOTYPE( void get_boot_parameters, (void)                    );
FORWARD _PROTOTYPE( void get_work, (void)                               );
```

```
FORWARD _PROTOTYPE( void load_ram, (void)                               );
FORWARD _PROTOTYPE( void load_super, (Dev_t super_dev)                  );


/*===========================================================================*
 *                              main                                         *
 *===========================================================================*/
PUBLIC void main()
{
/* This is the main program of the file system.  The main loop consists of
 * three major activities: getting new work, processing the work, and sending
 * the reply.  This loop never terminates as long as the file system runs.
 */
  int error;

  fs_init();

  /* This is the main loop that gets work, processes it, and sends replies. */
  while (TRUE) {
        get_work();             /* sets who and fs_call */

        fp = &fproc[who];       /* pointer to proc table struct */
        super_user = (fp->fp_effuid == SU_UID ? TRUE : FALSE);   /* su? */
        dont_reply = FALSE;     /* in other words, do reply is default */

        /* Call the internal function that does the work. */
        if (fs_call < 0 || fs_call >= NCALLS)
                error = EBADCALL;
        else
                error = (*call_vector[fs_call])();

        /* Copy the results back to the user and send reply. */
        if (dont_reply) continue;
        reply(who, error);
        if (rdahed_inode != NIL_INODE) read_ahead(); /* do block read ahead */
  }
}


/*===========================================================================*
 *                              get_work                                     *
 *===========================================================================*/
PRIVATE void get_work()
{
  /* Normally wait for new input.  However, if 'reviving' is
   * nonzero, a suspended process must be awakened.
   */

  register struct fproc *rp;

  if (reviving != 0) {
        /* Revive a suspended process. */
        for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++)
                if (rp->fp_revived == REVIVING) {
                        who = (int)(rp - fproc);
                        fs_call = rp->fp_fd & BYTE;
                        fd = (rp->fp_fd >>8) & BYTE;
                        buffer = rp->fp_buffer;
                        nbytes = rp->fp_nbytes;
                        rp->fp_suspended = NOT_SUSPENDED; /*no longer hanging*/
```

```
                        rp->fp_revived = NOT_REVIVING;
                        reviving--;
                        return;
                }
        panic("get_work couldn't revive anyone", NO_NUM);
  }

  /* Normal case.  No one to revive. */
  if (receive(ANY, &m) != OK) panic("fs receive error", NO_NUM);

  who = m.m_source;
  fs_call = m.m_type;
}


/*===========================================================================*
 *                              reply                                        *
 *===========================================================================*/
PUBLIC void reply(whom, result)
int whom;                       /* process to reply to */
int result;                     /* result of the call (usually OK or error #) */
{
/* Send a reply to a user process. It may fail (if the process has just
 * been killed by a signal), so don't check the return code.  If the send
 * fails, just ignore it.
 */

  reply_type = result;
  send(whom, &m1);
}


/*===========================================================================*
 *                              fs_init                                      *
 *===========================================================================*/
PRIVATE void fs_init()
{
/* Initialize global variables, tables, etc. */

  register struct inode *rip;
  int i;
  message mess;

  /* The following initializations are needed to let dev_opcl succeed .*/
  fp = (struct fproc *) NULL;
  who = FS_PROC_NR;

  buf_pool();                   /* initialize buffer pool */
  get_boot_parameters();        /* get the parameters from the menu */
  load_ram();                   /* init RAM disk, load if it is root */
  load_super(ROOT_DEV);         /* load super block for root device */

  /* Initialize the 'fproc' fields for process 0 .. INIT. */
  for (i = 0; i <= LOW_USER; i+= 1) {
        if (i == FS_PROC_NR) continue;  /* do not initialize FS */
        fp = &fproc[i];
        rip = get_inode(ROOT_DEV, ROOT_INODE);
        fp->fp_rootdir = rip;
        dup_inode(rip);
        fp->fp_workdir = rip;
```

```
        fp->fp_realuid = (uid_t) SYS_UID;
        fp->fp_effuid = (uid_t) SYS_UID;
        fp->fp_realgid = (gid_t) SYS_GID;
        fp->fp_effgid = (gid_t) SYS_GID;
        fp->fp_umask = ~0;
  }

  /* Certain relations must hold for the file system to work at all. */
  if (SUPER_SIZE > BLOCK_SIZE) panic("SUPER_SIZE > BLOCK_SIZE", NO_NUM);
  if (BLOCK_SIZE % V2_INODE_SIZE != 0)  /* this checks V1_INODE_SIZE too */
        panic("BLOCK_SIZE % V2_INODE_SIZE != 0", NO_NUM);
  if (OPEN_MAX > 127) panic("OPEN_MAX > 127", NO_NUM);
  if (NR_BUFS < 6) panic("NR_BUFS < 6", NO_NUM);
  if (V1_INODE_SIZE != 32) panic("V1 inode size != 32", NO_NUM);
  if (V2_INODE_SIZE != 64) panic("V2 inode size != 64", NO_NUM);
  if (OPEN_MAX > 8 * sizeof(long)) panic("Too few bits in fp_cloexec", NO_NUM);

  /* Tell the memory task where my process table is for the sake of ps(1). */
  mess.m_type = DEV_IOCTL;
  mess.PROC_NR = FS_PROC_NR;
  mess.REQUEST = MIOCSPSINFO;
  mess.ADDRESS = (void *) fproc;
  (void) sendrec(MEM, &mess);
}


/*===========================================================================*
 *                              buf_pool                                     *
 *===========================================================================*/
PRIVATE void buf_pool()
{
/* Initialize the buffer pool. */

  register struct buf *bp;

  bufs_in_use = 0;
  front = &buf[0];
  rear = &buf[NR_BUFS - 1];

  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) {
        bp->b_blocknr = NO_BLOCK;
        bp->b_dev = NO_DEV;
        bp->b_next = bp + 1;
        bp->b_prev = bp - 1;
  }
  buf[0].b_prev = NIL_BUF;
  buf[NR_BUFS - 1].b_next = NIL_BUF;

  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++) bp->b_hash = bp->b_next;
  buf_hash[0] = front;
}


/*===========================================================================*
 *                              get_boot_parameters                          *
 *===========================================================================*/
PUBLIC struct bparam_s boot_parameters;

PRIVATE void get_boot_parameters()
{
```

```
/* Ask kernel for boot parameters. */

  m1.m_type = SYS_GBOOT;
  m1.PROC1 = FS_PROC_NR;
  m1.MEM_PTR = (char *) &boot_parameters;
  (void) sendrec(SYSTASK, &m1);
}


/*===========================================================================*
 *                              load_ram                                     *
 *===========================================================================*/
PRIVATE void load_ram()
{
/* If the root device is the RAM disk, copy the entire root image device
 * block-by-block to a RAM disk with the same size as the image.
 * Otherwise, just allocate a RAM disk with size given in the boot parameters.
 */

  register struct buf *bp, *bp1;
  long k_loaded, lcount;
  u32_t ram_size, fsmax;
  zone_t zones;
  struct super_block *sp, *dsp;
  block_t b;
  int major, task;
  message dev_mess;

  ram_size = boot_parameters.bp_ramsize;

  /* Open the root device. */
  major = (ROOT_DEV >> MAJOR) & BYTE;   /* major device nr */
  task = dmap[major].dmap_task;         /* device task nr */
  dev_mess.m_type = DEV_OPEN;           /* distinguish from close */
  dev_mess.DEVICE = ROOT_DEV;
  dev_mess.COUNT = R_BIT|W_BIT;
  (*dmap[major].dmap_open)(task, &dev_mess);
  if (dev_mess.REP_STATUS != OK) panic("Cannot open root device",NO_NUM);

  /* If the root device is the ram disk then fill it from the image device. */
  if (ROOT_DEV == DEV_RAM) {
        major = (IMAGE_DEV >> MAJOR) & BYTE;    /* major device nr */
        task = dmap[major].dmap_task;           /* device task nr */
        dev_mess.m_type = DEV_OPEN;             /* distinguish from close */
        dev_mess.DEVICE = IMAGE_DEV;
        dev_mess.COUNT = R_BIT;
        (*dmap[major].dmap_open)(task, &dev_mess);
        if (dev_mess.REP_STATUS != OK) panic("Cannot open root device", NO_NUM);

        /* Get size of RAM disk by reading root file system's super block. */
        sp = &super_block[0];
        sp->s_dev = IMAGE_DEV;
        if (read_super(sp) != OK) panic("Bad root file system", NO_NUM);

        lcount = sp->s_zones << sp->s_log_zone_size;    /* # blks on root dev*/

        /* Stretch the RAM disk file system to the boot parameters size, but
         * no further than the last zone bit map block allows.
         */
        if (ram_size < lcount) ram_size = lcount;
```

```
          fsmax = (u32_t) sp->s_zmap_blocks * CHAR_BIT * BLOCK_SIZE;
          fsmax = (fsmax + (sp->s_firstdatazone-1)) << sp->s_log_zone_size;
          if (ram_size > fsmax) ram_size = fsmax;
    }

    /* Tell RAM driver how big the RAM disk must be. */
    m1.m_type = DEV_IOCTL;
    m1.PROC_NR = FS_PROC_NR;
    m1.REQUEST = MIOCRAMSIZE;
    m1.POSITION = ram_size;
    if (sendrec(MEM, &m1) != OK || m1.REP_STATUS != OK)
          panic("Can't set RAM disk size", NO_NUM);

    /* Tell MM the RAM disk size, and wait for it to come "on-line". */
    m1.m1_i1 = ((long) ram_size * BLOCK_SIZE) >> CLICK_SHIFT;
    if (sendrec(MM_PROC_NR, &m1) != OK)
          panic("FS can't sync up with MM", NO_NUM);

    /* If the root device is not the RAM disk, it doesn't need loading. */
    if (ROOT_DEV != DEV_RAM) return;

    /* Copy the blocks one at a time from the image to the RAM disk. */
    printf("Loading RAM disk.\33[23CLoaded:    0K ");

    inode[0].i_mode = I_BLOCK_SPECIAL;    /* temp inode for rahead() */
    inode[0].i_size = LONG_MAX;
    inode[0].i_dev = IMAGE_DEV;
    inode[0].i_zone[0] = IMAGE_DEV;

    for (b = 0; b < (block_t) lcount; b++) {
          bp = rahead(&inode[0], b, (off_t)BLOCK_SIZE * b, BLOCK_SIZE);
          bp1 = get_block(ROOT_DEV, b, NO_READ);
          memcpy(bp1->b_data, bp->b_data, (size_t) BLOCK_SIZE);
          bp1->b_dirt = DIRTY;
          put_block(bp, FULL_DATA_BLOCK);
          put_block(bp1, FULL_DATA_BLOCK);
          k_loaded = ( (long) b * BLOCK_SIZE)/1024L;      /* K loaded so far */
          if (k_loaded % 5 == 0) printf("\b\b\b\b\b\b\b%5ldK ", k_loaded);
    }

    printf("\rRAM disk loaded.\33[K\n\n");

    /* Close and invalidate image device. */
    dev_mess.m_type = DEV_CLOSE;
    dev_mess.DEVICE = IMAGE_DEV;
    (*dmap[major].dmap_close)(task, &dev_mess);
    invalidate(IMAGE_DEV);

    /* Resize the RAM disk root file system. */
    bp = get_block(ROOT_DEV, SUPER_BLOCK, NORMAL);
    dsp = (struct super_block *) bp->b_data;
    zones = ram_size >> sp->s_log_zone_size;
    dsp->s_nzones = conv2(sp->s_native, (u16_t) zones);
    dsp->s_zones = conv4(sp->s_native, zones);
    bp->b_dirt = DIRTY;
    put_block(bp, ZUPER_BLOCK);
  }


  /*===========================================================================*
```

```
 *                              load_super                                    *
 *===========================================================================*/
PRIVATE void load_super(super_dev)
dev_t super_dev;                        /* place to get superblock from */
{
  int bad;
  register struct super_block *sp;
  register struct inode *rip;

  /* Initialize the super_block table. */
  for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++)
        sp->s_dev = NO_DEV;

  /* Read in super_block for the root file system. */
  sp = &super_block[0];
  sp->s_dev = super_dev;

  /* Check super_block for consistency (is it the right diskette?). */
  bad = (read_super(sp) != OK);
  if (!bad) {
        rip = get_inode(super_dev, ROOT_INODE); /* inode for root dir */
        if ( (rip->i_mode & I_TYPE) != I_DIRECTORY || rip->i_nlinks < 3) bad++;
  }
  if (bad)panic("Invalid root file system.  Possibly wrong diskette.",NO_NUM);

  sp->s_imount = rip;
  dup_inode(rip);
  sp->s_isup = rip;
  sp->s_rd_only = 0;
  return;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/open.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the procedures for creating, opening, closing, and
 * seeking on files.
 *
 * The entry points into this file are
 *   do_creat:  perform the CREAT system call
 *   do_open:   perform the OPEN system call
 *   do_mknod:  perform the MKNOD system call
 *   do_mkdir:  perform the MKDIR system call
 *   do_close:  perform the CLOSE system call
 *   do_lseek:  perform the LSEEK system call
 */

#include "fs.h"
#include <sys/stat.h>
#include <fcntl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
```

```c
#include "fproc.h"
#include "inode.h"
#include "lock.h"
#include "param.h"

PRIVATE message dev_mess;
PRIVATE char mode_map[] = {R_BIT, W_BIT, R_BIT|W_BIT, 0};

FORWARD _PROTOTYPE( int common_open, (int oflags, Mode_t omode)         );
FORWARD _PROTOTYPE( int pipe_open, (struct inode *rip,Mode_t bits,int oflags));
FORWARD _PROTOTYPE( struct inode *new_node, (char *path, Mode_t bits,
                                                        zone_t z0)      );


/*===========================================================================*
 *                              do_creat                                     *
 *===========================================================================*/
PUBLIC int do_creat()
{
/* Perform the creat(name, mode) system call. */
  int r;

  if (fetch_name(name, name_length, M3) != OK) return(err_code);
  r = common_open(O_WRONLY | O_CREAT | O_TRUNC, (mode_t) mode);
  return(r);
}


/*===========================================================================*
 *                              do_open                                      *
 *===========================================================================*/
PUBLIC int do_open()
{
/* Perform the open(name, flags,...) system call. */

  int create_mode = 0;          /* is really mode_t but this gives problems */
  int r;

  /* If O_CREAT is set, open has three parameters, otherwise two. */
  if (mode & O_CREAT) {
        create_mode = c_mode;
        r = fetch_name(c_name, name1_length, M1);
  } else {
        r = fetch_name(name, name_length, M3);
  }

  if (r != OK) return(err_code); /* name was bad */
  r = common_open(mode, create_mode);
  return(r);
}


/*===========================================================================*
 *                              common_open                                  *
 *===========================================================================*/
PRIVATE int common_open(oflags, omode)
register int oflags;
mode_t omode;
{
/* Common code from do_creat and do_open. */
```

```

  register struct inode *rip;
  int r, b, major, task, exist = TRUE;
  dev_t dev;
  mode_t bits;
  off_t pos;
  struct filp *fil_ptr, *filp2;

  /* Remap the bottom two bits of oflags. */
  bits = (mode_t) mode_map[oflags & O_ACCMODE];

  /* See if file descriptor and filp slots are available. */
  if ( (r = get_fd(0, bits, &fd, &fil_ptr)) != OK) return(r);

  /* If O_CREATE is set, try to make the file. */
  if (oflags & O_CREAT) {
        /* Create a new inode by calling new_node(). */
        omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
        rip = new_node(user_path, omode, NO_ZONE);
        r = err_code;
        if (r == OK) exist = FALSE;      /* we just created the file */
        else if (r != EEXIST) return(r); /* other error */
        else exist = !(oflags & O_EXCL); /* file exists, if the O_EXCL
                                            flag is set this is an error */
  } else {
         /* Scan path name. */
        if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
  }

  /* Claim the file descriptor and filp slot and fill them in. */
  fp->fp_filp[fd] = fil_ptr;
  fil_ptr->filp_count = 1;
  fil_ptr->filp_ino = rip;
  fil_ptr->filp_flags = oflags;

  /* Only do the normal open code if we didn't just create the file. */
  if (exist) {
        /* Check protections. */
        if ((r = forbidden(rip, bits)) == OK) {
                /* Opening reg. files directories and special files differ. */
                switch (rip->i_mode & I_TYPE) {
                   case I_REGULAR:
                        /* Truncate regular file if O_TRUNC. */
                        if (oflags & O_TRUNC) {
                                if ((r = forbidden(rip, W_BIT)) !=OK) break;
                                truncate(rip);
                                wipe_inode(rip);
                                /* Send the inode from the inode cache to the
                                 * block cache, so it gets written on the next
                                 * cache flush.
                                 */
                                rw_inode(rip, WRITING);
                        }
                        break;

                   case I_DIRECTORY:
                        /* Directories may be read but not written. */
                        r = (bits & W_BIT ? EISDIR : OK);
                        break;

```

```
                     case I_CHAR_SPECIAL:
                     case I_BLOCK_SPECIAL:
                          /* Invoke the driver for special processing. */
                          dev_mess.m_type = DEV_OPEN;
                          dev = (dev_t) rip->i_zone[0];
                          dev_mess.DEVICE = dev;
                          dev_mess.COUNT = bits | (oflags & ~O_ACCMODE);
                          major = (dev >> MAJOR) & BYTE;  /* major device nr */
                          if (major <= 0 || major >= max_major) {
                                  r = ENODEV;
                                  break;
                          }
                          task = dmap[major].dmap_task;   /* device task nr */
                          (*dmap[major].dmap_open)(task, &dev_mess);
                          r = dev_mess.REP_STATUS;
                          break;

                     case I_NAMED_PIPE:
                          oflags |= O_APPEND;     /* force append mode */
                          fil_ptr->filp_flags = oflags;
                          r = pipe_open(rip, bits, oflags);
                          if (r == OK) {
                                  /* See if someone else is doing a rd or wt on
                                   * the FIFO.  If so, use its filp entry so the
                                   * file position will be automatically shared.
                                   */
                                  b = (bits & R_BIT ? R_BIT : W_BIT);
                                  fil_ptr->filp_count = 0; /* don't find self */
                                  if ((filp2 = find_filp(rip, b)) != NIL_FILP) {
                                          /* Co-reader or writer found. Use it.*/
                                          fp->fp_filp[fd] = filp2;
                                          filp2->filp_count++;
                                          filp2->filp_ino = rip;
                                          filp2->filp_flags = oflags;

                                          /* i_count was incremented incorrectly
                                           * by eatpath above, not knowing that
                                           * we were going to use an existing
                                           * filp entry.  Correct this error.
                                           */
                                          rip->i_count--;
                                  } else {
                                          /* Nobody else found.  Restore filp. */
                                          fil_ptr->filp_count = 1;
                                          if (b == R_BIT)
                                               pos = rip->i_zone[V2_NR_DZONES+1];
                                          else
                                               pos = rip->i_zone[V2_NR_DZONES+2];
                                          fil_ptr->filp_pos = pos;
                                  }
                          }
                          break;
                  }
          }
    }

    /* If error, release inode. */
    if (r != OK) {
          fp->fp_filp[fd] = NIL_FILP;
          fil_ptr->filp_count= 0;
```

```
        put_inode(rip);
        return(r);
  }

  return(fd);
}


/*===========================================================================*
 *                              new_node                                     *
 *===========================================================================*/
PRIVATE struct inode *new_node(path, bits, z0)
char *path;                     /* pointer to path name */
mode_t bits;                    /* mode of the new inode */
zone_t z0;                      /* zone number 0 for new inode */
{
/* New_node() is called by common_open(), do_mknod(), and do_mkdir().
 * In all cases it allocates a new inode, makes a directory entry for it on
 * the path 'path', and initializes it.  It returns a pointer to the inode if
 * it can do this; otherwise it returns NIL_INODE.  It always sets 'err_code'
 * to an appropriate value (OK or an error code).
 */

  register struct inode *rlast_dir_ptr, *rip;
  register int r;
  char string[NAME_MAX];

  /* See if the path can be opened down to the last directory. */
  if ((rlast_dir_ptr = last_dir(path, string)) == NIL_INODE) return(NIL_INODE);

  /* The final directory is accessible. Get final component of the path. */
  rip = advance(rlast_dir_ptr, string);
  if ( rip == NIL_INODE && err_code == ENOENT) {
        /* Last path component does not exist.  Make new directory entry. */
        if ( (rip = alloc_inode(rlast_dir_ptr->i_dev, bits)) == NIL_INODE) {
                /* Can't creat new inode: out of inodes. */
                put_inode(rlast_dir_ptr);
                return(NIL_INODE);
        }

        /* Force inode to the disk before making directory entry to make
         * the system more robust in the face of a crash: an inode with
         * no directory entry is much better than the opposite.
         */
        rip->i_nlinks++;
        rip->i_zone[0] = z0;            /* major/minor device numbers */
        rw_inode(rip, WRITING);         /* force inode to disk now */

        /* New inode acquired.  Try to make directory entry. */
        if ((r = search_dir(rlast_dir_ptr, string, &rip->i_num,ENTER)) != OK) {
                put_inode(rlast_dir_ptr);
                rip->i_nlinks--;        /* pity, have to free disk inode */
                rip->i_dirt = DIRTY;    /* dirty inodes are written out */
                put_inode(rip); /* this call frees the inode */
                err_code = r;
                return(NIL_INODE);
        }

  } else {
        /* Either last component exists, or there is some problem. */
```

```
        if (rip != NIL_INODE)
                r = EEXIST;
        else
                r = err_code;
  }

  /* Return the directory inode and exit. */
  put_inode(rlast_dir_ptr);
  err_code = r;
  return(rip);
}


/*===========================================================================*
 *                              pipe_open                                    *
 *===========================================================================*/
PRIVATE int pipe_open(rip, bits, oflags)
register struct inode *rip;
register mode_t bits;
register int oflags;
{
/*  This function is called from common_open. It checks if
 *  there is at least one reader/writer pair for the pipe, if not
 *  it suspends the caller, otherwise it revives all other blocked
 *  processes hanging on the pipe.
 */

  if (find_filp(rip, bits & W_BIT ? R_BIT : W_BIT) == NIL_FILP) {
        if (oflags & O_NONBLOCK) {
                if (bits & W_BIT) return(ENXIO);
        } else
                suspend(XPOPEN);        /* suspend caller */
  } else if (susp_count > 0) {/* revive blocked processes */
        release(rip, OPEN, susp_count);
        release(rip, CREAT, susp_count);
  }
  rip->i_pipe = I_PIPE;

  return(OK);
}


/*===========================================================================*
 *                              do_mknod                                     *
 *===========================================================================*/
PUBLIC int do_mknod()
{
/* Perform the mknod(name, mode, addr) system call. */

  register mode_t bits, mode_bits;
  struct inode *ip;

  /* Only the super_user may make nodes other than fifos. */
  mode_bits = (mode_t) m.m1_i2; /* mode of the inode */
  if (!super_user && ((mode_bits & I_TYPE) != I_NAMED_PIPE)) return(EPERM);
  if (fetch_name(m.m1_p1, m.m1_i1, M1) != OK) return(err_code);
  bits = (mode_bits & I_TYPE) | (mode_bits & ALL_MODES & fp->fp_umask);
  ip = new_node(user_path, bits, (zone_t) m.m1_i3);
  put_inode(ip);
  return(err_code);
```

```
}


/*===========================================================================*
 *                              do_mkdir                                     *
 *===========================================================================*/
PUBLIC int do_mkdir()
{
/* Perform the mkdir(name, mode) system call. */

  int r1, r2;                   /* status codes */
  ino_t dot, dotdot;            /* inode numbers for . and .. */
  mode_t bits;                  /* mode bits for the new inode */
  char string[NAME_MAX];        /* last component of the new dir's path name */
  register struct inode *rip, *ldirp;

  /* Check to see if it is possible to make another link in the parent dir. */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  ldirp = last_dir(user_path, string);  /* pointer to new dir's parent */
  if (ldirp == NIL_INODE) return(err_code);
  if ( (ldirp->i_nlinks & BYTE) >= LINK_MAX) {
        put_inode(ldirp);       /* return parent */
        return(EMLINK);
  }

  /* Next make the inode. If that fails, return error code. */
  bits = I_DIRECTORY | (mode & RWX_MODES & fp->fp_umask);
  rip = new_node(user_path, bits, (zone_t) 0);
  if (rip == NIL_INODE || err_code == EEXIST) {
        put_inode(rip);         /* can't make dir: it already exists */
        put_inode(ldirp);       /* return parent too */
        return(err_code);
  }

  /* Get the inode numbers for . and .. to enter in the directory. */
  dotdot = ldirp->i_num;        /* parent's inode number */
  dot = rip->i_num;             /* inode number of the new dir itself */

  /* Now make dir entries for . and .. unless the disk is completely full. */
  /* Use dot1 and dot2, so the mode of the directory isn't important. */
  rip->i_mode = bits;   /* set mode */
  r1 = search_dir(rip, dot1, &dot, ENTER);      /* enter . in the new dir */
  r2 = search_dir(rip, dot2, &dotdot, ENTER);   /* enter .. in the new dir */

  /* If both . and .. were successfully entered, increment the link counts. */
  if (r1 == OK && r2 == OK) {
        /* Normal case.  It was possible to enter . and .. in the new dir. */
        rip->i_nlinks++;        /* this accounts for . */
        ldirp->i_nlinks++;      /* this accounts for .. */
        ldirp->i_dirt = DIRTY;  /* mark parent's inode as dirty */
  } else {
        /* It was not possible to enter . or .. probably disk was full. */
        (void) search_dir(ldirp, string, (ino_t *) 0, DELETE);
        rip->i_nlinks--;        /* undo the increment done in new_node() */
  }
  rip->i_dirt = DIRTY;          /* either way, i_nlinks has changed */

  put_inode(ldirp);             /* return the inode of the parent dir */
  put_inode(rip);               /* return the inode of the newly made dir */
  return(err_code);             /* new_node() always sets 'err_code' */
```

```
}


/*===========================================================================*
 *                              do_close                                     *
 *===========================================================================*/
PUBLIC int do_close()
{
/* Perform the close(fd) system call. */

  register struct filp *rfilp;
  register struct inode *rip;
  struct file_lock *flp;
  int rw, mode_word, major, task, lock_count;
  dev_t dev;

  /* First locate the inode that belongs to the file descriptor. */
  if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);
  rip = rfilp->filp_ino;        /* 'rip' points to the inode */

  if (rfilp->filp_count - 1 == 0 && rfilp->filp_mode != FILP_CLOSED) {
        /* Check to see if the file is special. */
        mode_word = rip->i_mode & I_TYPE;
        if (mode_word == I_CHAR_SPECIAL || mode_word == I_BLOCK_SPECIAL) {
                dev = (dev_t) rip->i_zone[0];
                if (mode_word == I_BLOCK_SPECIAL)  {
                        /* Invalidate cache entries unless special is mounted
                         * or ROOT
                         */
                        if (!mounted(rip)) {
                                (void) do_sync();       /* purge cache */
                                invalidate(dev);
                        }
                }
                /* Use the dmap_close entry to do any special processing
                 * required.
                 */
                dev_mess.m_type = DEV_CLOSE;
                dev_mess.DEVICE = dev;
                major = (dev >> MAJOR) & BYTE;  /* major device nr */
                task = dmap[major].dmap_task;   /* device task nr */
                (*dmap[major].dmap_close)(task, &dev_mess);
        }
  }

  /* If the inode being closed is a pipe, release everyone hanging on it. */
  if (rip->i_pipe == I_PIPE) {
        rw = (rfilp->filp_mode & R_BIT ? WRITE : READ);
        release(rip, rw, NR_PROCS);
  }

  /* If a write has been done, the inode is already marked as DIRTY. */
  if (--rfilp->filp_count == 0) {
        if (rip->i_pipe == I_PIPE && rip->i_count > 1) {
                /* Save the file position in the i-node in case needed later.
                 * The read and write positions are saved separately.  The
                 * last 3 zones in the i-node are not used for (named) pipes.
                 */
                if (rfilp->filp_mode == R_BIT)
                        rip->i_zone[V2_NR_DZONES+1] = (zone_t) rfilp->filp_pos;
```

```
                else
                        rip->i_zone[V2_NR_DZONES+2] = (zone_t) rfilp->filp_pos;
        }
        put_inode(rip);
  }

  fp->fp_cloexec &= ~(1L << fd);        /* turn off close-on-exec bit */
  fp->fp_filp[fd] = NIL_FILP;

  /* Check to see if the file is locked.  If so, release all locks. */
  if (nr_locks == 0) return(OK);
  lock_count = nr_locks;        /* save count of locks */
  for (flp = &file_lock[0]; flp < &file_lock[NR_LOCKS]; flp++) {
        if (flp->lock_type == 0) continue;      /* slot not in use */
        if (flp->lock_inode == rip && flp->lock_pid == fp->fp_pid) {
                flp->lock_type = 0;
                nr_locks--;
        }
  }
  if (nr_locks < lock_count) lock_revive();     /* lock released */
  return(OK);
}


/*===========================================================================*
 *                              do_lseek                                     *
 *===========================================================================*/
PUBLIC int do_lseek()
{
/* Perform the lseek(ls_fd, offset, whence) system call. */

  register struct filp *rfilp;
  register off_t pos;

  /* Check to see if the file descriptor is valid. */
  if ( (rfilp = get_filp(ls_fd)) == NIL_FILP) return(err_code);

  /* No lseek on pipes. */
  if (rfilp->filp_ino->i_pipe == I_PIPE) return(ESPIPE);

  /* The value of 'whence' determines the start position to use. */
  switch(whence) {
        case 0: pos = 0;        break;
        case 1: pos = rfilp->filp_pos;  break;
        case 2: pos = rfilp->filp_ino->i_size;  break;
        default: return(EINVAL);
  }

  /* Check for overflow. */
  if (((long)offset > 0) && ((long)(pos + offset) < (long)pos)) return(EINVAL);
  if (((long)offset < 0) && ((long)(pos + offset) > (long)pos)) return(EINVAL);
  pos = pos + offset;

  if (pos != rfilp->filp_pos)
        rfilp->filp_ino->i_seek = ISEEK;        /* inhibit read ahead */
  rfilp->filp_pos = pos;
  reply_l1 = pos;               /* insert the long into the output message */
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/read.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains the heart of the mechanism used to read (and write)
 * files.  Read and write requests are split up into chunks that do not cross
 * block boundaries.  Each chunk is then processed in turn.  Reads on special
 * files are also detected and handled.
 *
 * The entry points into this file are
 *   do_read:    perform the READ system call by calling read_write
 *   read_write: actually do the work of READ and WRITE
 *   read_map:   given an inode and file position, look up its zone number
 *   rd_indir:   read an entry in an indirect block
 *   read_ahead: manage the block read ahead business
 */

#include "fs.h"
#include <fcntl.h>
#include <minix/com.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

#define FD_MASK          077    /* max file descriptor is 63 */

PRIVATE message umess;          /* message for asking SYSTASK for user copy */

FORWARD _PROTOTYPE( int rw_chunk, (struct inode *rip, off_t position,
                        unsigned off, int chunk, unsigned left, int rw_flag,
                        char *buff, int seg, int usr)                   );

/*===========================================================================*
 *                              do_read                                      *
 *===========================================================================*/
PUBLIC int do_read()
{
  return(read_write(READING));
}


/*===========================================================================*
 *                              read_write                                   *
 *===========================================================================*/
PUBLIC int read_write(rw_flag)
int rw_flag;                    /* READING or WRITING */
{
/* Perform read(fd, buffer, nbytes) or write(fd, buffer, nbytes) call. */

  register struct inode *rip;
  register struct filp *f;
  off_t bytes_left, f_size, position;
  unsigned int off, cum_io;
  int op, oflags, r, chunk, usr, seg, block_spec, char_spec;
  int regular, partial_pipe = 0, partial_cnt = 0;
  dev_t dev;
```

```
  mode_t mode_word;
  struct filp *wf;

  /* MM loads segments by putting funny things in upper 10 bits of 'fd'. */
  if (who == MM_PROC_NR && (fd & (~BYTE)) ) {
        usr = (fd >> 8) & BYTE;
        seg = (fd >> 6) & 03;
        fd &= FD_MASK;            /* get rid of user and segment bits */
  } else {
        usr = who;                /* normal case */
        seg = D;
  }

  /* If the file descriptor is valid, get the inode, size and mode. */
  if (nbytes < 0) return(EINVAL);
  if ((f = get_filp(fd)) == NIL_FILP) return(err_code);
  if (((f->filp_mode) & (rw_flag == READING ? R_BIT : W_BIT)) == 0) {
        return(f->filp_mode == FILP_CLOSED ? EIO : EBADF);
  }
  if (nbytes == 0) return(0);   /* so char special files need not check for 0*/
  position = f->filp_pos;
  if (position > MAX_FILE_POS) return(EINVAL);
  if (position + nbytes < position) return(EINVAL); /* unsigned overflow */
  oflags = f->filp_flags;
  rip = f->filp_ino;
  f_size = rip->i_size;
  r = OK;
  if (rip->i_pipe == I_PIPE) {
        /* fp->fp_cum_io_partial is only nonzero when doing partial writes */
        cum_io = fp->fp_cum_io_partial;
  } else {
        cum_io = 0;
  }
  op = (rw_flag == READING ? DEV_READ : DEV_WRITE);
  mode_word = rip->i_mode & I_TYPE;
  regular = mode_word == I_REGULAR || mode_word == I_NAMED_PIPE;

  char_spec = (mode_word == I_CHAR_SPECIAL ? 1 : 0);
  block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);
  if (block_spec) f_size = LONG_MAX;
  rdwt_err = OK;                /* set to EIO if disk error occurs */

  /* Check for character special files. */
  if (char_spec) {
        dev = (dev_t) rip->i_zone[0];
        r = dev_io(op, oflags & O_NONBLOCK, dev, position, nbytes, who,buffer);
        if (r >= 0) {
                cum_io = r;
                position += r;
                r = OK;
        }
  } else {
        if (rw_flag == WRITING && block_spec == 0) {
                /* Check in advance to see if file will grow too big. */
                if (position > rip->i_sp->s_max_size - nbytes) return(EFBIG);

                /* Check for O_APPEND flag. */
                if (oflags & O_APPEND) position = f_size;

                /* Clear the zone containing present EOF if hole about
```

```
               * to be created.  This is necessary because all unwritten
               * blocks prior to the EOF must read as zeros.
               */
              if (position > f_size) clear_zone(rip, f_size, 0);
      }

      /* Pipes are a little different.  Check. */
      if (rip->i_pipe == I_PIPE) {
             r = pipe_check(rip,rw_flag,oflags,nbytes,position,&partial_cnt);
             if (r <= 0) return(r);
      }

      if (partial_cnt > 0) partial_pipe = 1;

      /* Split the transfer into chunks that don't span two blocks. */
      while (nbytes != 0) {
              off = (unsigned int) (position % BLOCK_SIZE);/* offset in blk*/
              if (partial_pipe) {  /* pipes only */
                      chunk = MIN(partial_cnt, BLOCK_SIZE - off);
              } else
                      chunk = MIN(nbytes, BLOCK_SIZE - off);
              if (chunk < 0) chunk = BLOCK_SIZE - off;

              if (rw_flag == READING) {
                      bytes_left = f_size - position;
                      if (position >= f_size) break;  /* we are beyond EOF */
                      if (chunk > bytes_left) chunk = (int) bytes_left;
              }

              /* Read or write 'chunk' bytes. */
              r = rw_chunk(rip, position, off, chunk, (unsigned) nbytes,
                           rw_flag, buffer, seg, usr);
              if (r != OK) break;     /* EOF reached */
              if (rdwt_err < 0) break;

              /* Update counters and pointers. */
              buffer += chunk;        /* user buffer address */
              nbytes -= chunk;        /* bytes yet to be read */
              cum_io += chunk;        /* bytes read so far */
              position += chunk;      /* position within the file */

              if (partial_pipe) {
                      partial_cnt -= chunk;
                      if (partial_cnt <= 0)  break;
              }
      }
}

/* On write, update file size and access time. */
if (rw_flag == WRITING) {
      if (regular || mode_word == I_DIRECTORY) {
              if (position > f_size) rip->i_size = position;
      }
} else {
      if (rip->i_pipe == I_PIPE && position >= rip->i_size) {
              /* Reset pipe pointers. */
              rip->i_size = 0;        /* no data left */
              position = 0;           /* reset reader(s) */
              if ( (wf = find_filp(rip, W_BIT)) != NIL_FILP) wf->filp_pos =0;
      }
```

```
  }
  f->filp_pos = position;

  /* Check to see if read-ahead is called for, and if so, set it up. */
  if (rw_flag == READING && rip->i_seek == NO_SEEK && position % BLOCK_SIZE== 0
              && (regular || mode_word == I_DIRECTORY)) {
        rdahed_inode = rip;
        rdahedpos = position;
  }
  rip->i_seek = NO_SEEK;

  if (rdwt_err != OK) r = rdwt_err;     /* check for disk error */
  if (rdwt_err == END_OF_FILE) r = OK;
  if (r == OK) {
        if (rw_flag == READING) rip->i_update |= ATIME;
        if (rw_flag == WRITING) rip->i_update |= CTIME | MTIME;
        rip->i_dirt = DIRTY;            /* inode is thus now dirty */
        if (partial_pipe) {
                partial_pipe = 0;
                        /* partial write on pipe with */
                        /* O_NONBLOCK, return write count */
                if (!(oflags & O_NONBLOCK)) {
                        fp->fp_cum_io_partial = cum_io;
                        suspend(XPIPE); /* partial write on pipe with */
                        return(0);      /* nbyte > PIPE_SIZE - non-atomic */
                }
        }
        fp->fp_cum_io_partial = 0;
        return(cum_io);
  } else {
        return(r);
  }
}


/*===========================================================================*
 *                              rw_chunk                                     *
 *===========================================================================*/
PRIVATE int rw_chunk(rip, position, off, chunk, left, rw_flag, buff, seg, usr)
register struct inode *rip;     /* pointer to inode for file to be rd/wr */
off_t position;                 /* position within file to read or write */
unsigned off;                   /* off within the current block */
int chunk;                      /* number of bytes to read or write */
unsigned left;                  /* max number of bytes wanted after position */
int rw_flag;                    /* READING or WRITING */
char *buff;                     /* virtual address of the user buffer */
int seg;                        /* T or D segment in user space */
int usr;                        /* which user process */
{
/* Read or write (part of) a block. */

  register struct buf *bp;
  register int r;
  int n, block_spec;
  block_t b;
  dev_t dev;

  block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
  if (block_spec) {
        b = position/BLOCK_SIZE;
```

```c
        dev = (dev_t) rip->i_zone[0];
  } else {
        b = read_map(rip, position);
        dev = rip->i_dev;
  }

  if (!block_spec && b == NO_BLOCK) {
        if (rw_flag == READING) {
                /* Reading from a nonexistent block.  Must read as all zeros.*/
                bp = get_block(NO_DEV, NO_BLOCK, NORMAL);    /* get a buffer */
                zero_block(bp);
        } else {
                /* Writing to a nonexistent block. Create and enter in inode.*/
                if ((bp= new_block(rip, position)) == NIL_BUF)return(err_code);
        }
  } else if (rw_flag == READING) {
        /* Read and read ahead if convenient. */
        bp = rahead(rip, b, position, left);
  } else {
        /* Normally an existing block to be partially overwritten is first read
         * in.  However, a full block need not be read in.  If it is already in
         * the cache, acquire it, otherwise just acquire a free buffer.
         */
        n = (chunk == BLOCK_SIZE ? NO_READ : NORMAL);
        if (!block_spec && off == 0 && position >= rip->i_size) n = NO_READ;
        bp = get_block(dev, b, n);
  }

  /* In all cases, bp now points to a valid buffer. */
  if (rw_flag == WRITING && chunk != BLOCK_SIZE && !block_spec &&
                                        position >= rip->i_size && off == 0) {
        zero_block(bp);
  }
  if (rw_flag == READING) {
        /* Copy a chunk from the block buffer to user space. */
        r = sys_copy(FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
                        usr, seg, (phys_bytes) buff,
                        (phys_bytes) chunk);
  } else {
        /* Copy a chunk from user space to the block buffer. */
        r = sys_copy(usr, seg, (phys_bytes) buff,
                        FS_PROC_NR, D, (phys_bytes) (bp->b_data+off),
                        (phys_bytes) chunk);
        bp->b_dirt = DIRTY;
  }
  n = (off + chunk == BLOCK_SIZE ? FULL_DATA_BLOCK : PARTIAL_DATA_BLOCK);
  put_block(bp, n);
  return(r);
}


/*===========================================================================*
 *                              read_map                                     *
 *===========================================================================*/
PUBLIC block_t read_map(rip, position)
register struct inode *rip;     /* ptr to inode to map from */
off_t position;                 /* position in file whose blk wanted */
{
/* Given an inode and a position within the corresponding file, locate the
 * block (not zone) number in which that position is to be found and return it.
```

```
 */

  register struct buf *bp;
  register zone_t z;
  int scale, boff, dzones, nr_indirects, index, zind, ex;
  block_t b;
  long excess, zone, block_pos;

  scale = rip->i_sp->s_log_zone_size;   /* for block-zone conversion */
  block_pos = position/BLOCK_SIZE;      /* relative blk # in file */
  zone = block_pos >> scale;    /* position's zone */
  boff = (int) (block_pos - (zone << scale) ); /* relative blk # within zone */
  dzones = rip->i_ndzones;
  nr_indirects = rip->i_nindirs;

  /* Is 'position' to be found in the inode itself? */
  if (zone < dzones) {
        zind = (int) zone;      /* index should be an int */
        z = rip->i_zone[zind];
        if (z == NO_ZONE) return(NO_BLOCK);
        b = ((block_t) z << scale) + boff;
        return(b);
  }

  /* It is not in the inode, so it must be single or double indirect. */
  excess = zone - dzones;       /* first Vx_NR_DZONES don't count */

  if (excess < nr_indirects) {
        /* 'position' can be located via the single indirect block. */
        z = rip->i_zone[dzones];
  } else {
        /* 'position' can be located via the double indirect block. */
        if ( (z = rip->i_zone[dzones+1]) == NO_ZONE) return(NO_BLOCK);
        excess -= nr_indirects;                 /* single indir doesn't count*/
        b = (block_t) z << scale;
        bp = get_block(rip->i_dev, b, NORMAL);  /* get double indirect block */
        index = (int) (excess/nr_indirects);
        z = rd_indir(bp, index);                /* z= zone for single*/
        put_block(bp, INDIRECT_BLOCK);          /* release double ind block */
        excess = excess % nr_indirects;         /* index into single ind blk */
  }

  /* 'z' is zone num for single indirect block; 'excess' is index into it. */
  if (z == NO_ZONE) return(NO_BLOCK);
  b = (block_t) z << scale;                     /* b is blk # for single ind */
  bp = get_block(rip->i_dev, b, NORMAL);        /* get single indirect block */
  ex = (int) excess;                            /* need an integer */
  z = rd_indir(bp, ex);                         /* get block pointed to */
  put_block(bp, INDIRECT_BLOCK);                /* release single indir blk */
  if (z == NO_ZONE) return(NO_BLOCK);
  b = ((block_t) z << scale) + boff;
  return(b);
}


/*===========================================================================*
 *                              rd_indir                                     *
 *===========================================================================*/
PUBLIC zone_t rd_indir(bp, index)
struct buf *bp;                 /* pointer to indirect block */
```

```
int index;                      /* index into *bp */
{
/* Given a pointer to an indirect block, read one entry.  The reason for
 * making a separate routine out of this is that there are four cases:
 * V1 (IBM and 68000), and V2 (IBM and 68000).
 */

  struct super_block *sp;
  zone_t zone;                  /* V2 zones are longs (shorts in V1) */

  sp = get_super(bp->b_dev);    /* need super block to find file sys type */

  /* read a zone from an indirect block */
  if (sp->s_version == V1)
        zone = (zone_t) conv2(sp->s_native, (int)  bp->b_v1_ind[index]);
  else
        zone = (zone_t) conv4(sp->s_native, (long) bp->b_v2_ind[index]);

  if (zone != NO_ZONE &&
                (zone < (zone_t) sp->s_firstdatazone || zone >= sp->s_zones)) {
        printf("Illegal zone number %ld in indirect block, index %d\n",
               (long) zone, index);
        panic("check file system", NO_NUM);
  }
  return(zone);
}


/*===========================================================================*
 *                              read_ahead                                   *
 *===========================================================================*/
PUBLIC void read_ahead()
{
/* Read a block into the cache before it is needed. */

  register struct inode *rip;
  struct buf *bp;
  block_t b;

  rip = rdahed_inode;           /* pointer to inode to read ahead from */
  rdahed_inode = NIL_INODE;     /* turn off read ahead */
  if ( (b = read_map(rip, rdahedpos)) == NO_BLOCK) return;      /* at EOF */
  bp = rahead(rip, b, rdahedpos, BLOCK_SIZE);
  put_block(bp, PARTIAL_DATA_BLOCK);
}


/*===========================================================================*
 *                              rahead                                       *
 *===========================================================================*/
PUBLIC struct buf *rahead(rip, baseblock, position, bytes_ahead)
register struct inode *rip;     /* pointer to inode for file to be read */
block_t baseblock;              /* block at current position */
off_t position;                 /* position within file */
unsigned bytes_ahead;           /* bytes beyond position for immediate use */
{
/* Fetch a block from the cache or the device.  If a physical read is
 * required, prefetch as many more blocks as convenient into the cache.
 * This usually covers bytes_ahead and is at least BLOCKS_MINIMUM.
 * The device driver may decide it knows better and stop reading at a
```

```
   * cylinder boundary (or after an error).  Rw_scattered() puts an optional
   * flag on all reads to allow this.
   */

  /* Minimum number of blocks to prefetch. */
  # define BLOCKS_MINIMUM         (NR_BUFS < 50 ? 18 : 32)

    int block_spec, scale, read_q_size;
    unsigned int blocks_ahead, fragment;
    block_t block, blocks_left;
    off_t ind1_pos;
    dev_t dev;
    struct buf *bp;
    static struct buf *read_q[NR_BUFS];

    block_spec = (rip->i_mode & I_TYPE) == I_BLOCK_SPECIAL;
    if (block_spec) {
          dev = (dev_t) rip->i_zone[0];
    } else {
          dev = rip->i_dev;
    }

    block = baseblock;
    bp = get_block(dev, block, PREFETCH);
    if (bp->b_dev != NO_DEV) return(bp);

    /* The best guess for the number of blocks to prefetch:  A lot.
     * It is impossible to tell what the device looks like, so we don't even
     * try to guess the geometry, but leave it to the driver.
     *
     * The floppy driver can read a full track with no rotational delay, and it
     * avoids reading partial tracks if it can, so handing it enough buffers to
     * read two tracks is perfect.  (Two, because some diskette types have
     * an odd number of sectors per track, so a block may span tracks.)
     *
     * The disk drivers don't try to be smart.  With todays disks it is
     * impossible to tell what the real geometry looks like, so it is best to
     * read as much as you can.  With luck the caching on the drive allows
     * for a little time to start the next read.
     *
     * The current solution below is a bit of a hack, it just reads blocks from
     * the current file position hoping that more of the file can be found.  A
     * better solution must look at the already available zone pointers and
     * indirect blocks (but don't call read_map!).
     */

    fragment = position % BLOCK_SIZE;
    position -= fragment;
    bytes_ahead += fragment;

    blocks_ahead = (bytes_ahead + BLOCK_SIZE - 1) / BLOCK_SIZE;

    if (block_spec && rip->i_size == 0) {
          blocks_left = NR_IOREQS;
    } else {
          blocks_left = (rip->i_size - position + BLOCK_SIZE - 1) / BLOCK_SIZE;

          /* Go for the first indirect block if we are in its neighborhood. */
          if (!block_spec) {
                  scale = rip->i_sp->s_log_zone_size;
```

```
                ind1_pos = (off_t) rip->i_ndzones * (BLOCK_SIZE << scale);
                if (position <= ind1_pos && rip->i_size > ind1_pos) {
                        blocks_ahead++;
                        blocks_left++;
                }
        }
  }

  /* No more than the maximum request. */
  if (blocks_ahead > NR_IOREQS) blocks_ahead = NR_IOREQS;

  /* Read at least the minimum number of blocks, but not after a seek. */
  if (blocks_ahead < BLOCKS_MINIMUM && rip->i_seek == NO_SEEK)
        blocks_ahead = BLOCKS_MINIMUM;

  /* Can't go past end of file. */
  if (blocks_ahead > blocks_left) blocks_ahead = blocks_left;

  read_q_size = 0;

  /* Acquire block buffers. */
  for (;;) {
        read_q[read_q_size++] = bp;

        if (--blocks_ahead == 0) break;

        /* Don't trash the cache, leave 4 free. */
        if (bufs_in_use >= NR_BUFS - 4) break;

        block++;

        bp = get_block(dev, block, PREFETCH);
        if (bp->b_dev != NO_DEV) {
                /* Oops, block already in the cache, get out. */
                put_block(bp, FULL_DATA_BLOCK);
                break;
        }
  }
  rw_scattered(dev, read_q, read_q_size, READING);
  return(get_block(dev, baseblock, NORMAL));
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/write.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file is the counterpart of "read.c".  It contains the code for writing
 * insofar as this is not contained in read_write().
 *
 * The entry points into this file are
 *   do_write:     call read_write to perform the WRITE system call
 *   clear_zone:   erase a zone in the middle of a file
 *   new_block:    acquire a new block
 */

#include "fs.h"
```

```
#include <string.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

FORWARD _PROTOTYPE( int write_map, (struct inode *rip, off_t position,
                        zone_t new_zone)                                );

FORWARD _PROTOTYPE( void wr_indir, (struct buf *bp, int index, zone_t zone) );

/*===========================================================================*
 *                              do_write                                     *
 *===========================================================================*/
PUBLIC int do_write()
{
/* Perform the write(fd, buffer, nbytes) system call. */

  return(read_write(WRITING));
}


/*===========================================================================*
 *                              write_map                                    *
 *===========================================================================*/
PRIVATE int write_map(rip, position, new_zone)
register struct inode *rip;     /* pointer to inode to be changed */
off_t position;                 /* file address to be mapped */
zone_t new_zone;                /* zone # to be inserted */
{
/* Write a new zone into an inode. */
  int scale, ind_ex, new_ind, new_dbl, zones, nr_indirects, single, zindex, ex;
  zone_t z, z1;
  register block_t b;
  long excess, zone;
  struct buf *bp;

  rip->i_dirt = DIRTY;          /* inode will be changed */
  bp = NIL_BUF;
  scale = rip->i_sp->s_log_zone_size;           /* for zone-block conversion */
  zone = (position/BLOCK_SIZE) >> scale;        /* relative zone # to insert */
  zones = rip->i_ndzones;       /* # direct zones in the inode */
  nr_indirects = rip->i_nindirs;/* # indirect zones per indirect block */

  /* Is 'position' to be found in the inode itself? */
  if (zone < zones) {
        zindex = (int) zone;    /* we need an integer here */
        rip->i_zone[zindex] = new_zone;
        return(OK);
  }

  /* It is not in the inode, so it must be single or double indirect. */
  excess = zone - zones;        /* first Vx_NR_DZONES don't count */
  new_ind = FALSE;
  new_dbl = FALSE;

  if (excess < nr_indirects) {
        /* 'position' can be located via the single indirect block. */
        z1 = rip->i_zone[zones];        /* single indirect zone */
```

```c
        single = TRUE;
  } else {
        /* 'position' can be located via the double indirect block. */
        if ( (z = rip->i_zone[zones+1]) == NO_ZONE) {
                /* Create the double indirect block. */
                if ( (z = alloc_zone(rip->i_dev, rip->i_zone[0])) == NO_ZONE)
                        return(err_code);
                rip->i_zone[zones+1] = z;
                new_dbl = TRUE; /* set flag for later */
        }

        /* Either way, 'z' is zone number for double indirect block. */
        excess -= nr_indirects; /* single indirect doesn't count */
        ind_ex = (int) (excess / nr_indirects);
        excess = excess % nr_indirects;
        if (ind_ex >= nr_indirects) return(EFBIG);
        b = (block_t) z << scale;
        bp = get_block(rip->i_dev, b, (new_dbl ? NO_READ : NORMAL));
        if (new_dbl) zero_block(bp);
        z1 = rd_indir(bp, ind_ex);
        single = FALSE;
  }

  /* z1 is now single indirect zone; 'excess' is index. */
  if (z1 == NO_ZONE) {
        /* Create indirect block and store zone # in inode or dbl indir blk. */
        z1 = alloc_zone(rip->i_dev, rip->i_zone[0]);
        if (single)
                rip->i_zone[zones] = z1;        /* update inode */
        else
                wr_indir(bp, ind_ex, z1);       /* update dbl indir */

        new_ind = TRUE;
        if (bp != NIL_BUF) bp->b_dirt = DIRTY;  /* if double ind, it is dirty*/
        if (z1 == NO_ZONE) {
                put_block(bp, INDIRECT_BLOCK);  /* release dbl indirect blk */
                return(err_code);       /* couldn't create single ind */
        }
  }
  put_block(bp, INDIRECT_BLOCK);        /* release double indirect blk */

  /* z1 is indirect block's zone number. */
  b = (block_t) z1 << scale;
  bp = get_block(rip->i_dev, b, (new_ind ? NO_READ : NORMAL) );
  if (new_ind) zero_block(bp);
  ex = (int) excess;                    /* we need an int here */
  wr_indir(bp, ex, new_zone);
  bp->b_dirt = DIRTY;
  put_block(bp, INDIRECT_BLOCK);

  return(OK);
}


/*===========================================================================*
 *                              wr_indir                                     *
 *===========================================================================*/
PRIVATE void wr_indir(bp, index, zone)
struct buf *bp;                 /* pointer to indirect block */
int index;                      /* index into *bp */
```

```
zone_t zone;                    /* zone to write */
{
/* Given a pointer to an indirect block, write one entry. */

  struct super_block *sp;

  sp = get_super(bp->b_dev);    /* need super block to find file sys type */

  /* write a zone into an indirect block */
  if (sp->s_version == V1)
        bp->b_v1_ind[index] = (zone1_t) conv2(sp->s_native, (int)  zone);
  else
        bp->b_v2_ind[index] = (zone_t)  conv4(sp->s_native, (long) zone);
}


/*===========================================================================*
 *                              clear_zone                                   *
 *===========================================================================*/
PUBLIC void clear_zone(rip, pos, flag)
register struct inode *rip;     /* inode to clear */
off_t pos;                      /* points to block to clear */
int flag;                       /* 0 if called by read_write, 1 by new_block */
{
/* Zero a zone, possibly starting in the middle.  The parameter 'pos' gives
 * a byte in the first block to be zeroed.  Clearzone() is called from
 * read_write and new_block().
 */

  register struct buf *bp;
  register block_t b, blo, bhi;
  register off_t next;
  register int scale;
  register zone_t zone_size;

  /* If the block size and zone size are the same, clear_zone() not needed. */
  scale = rip->i_sp->s_log_zone_size;
  if (scale == 0) return;

  zone_size = (zone_t) BLOCK_SIZE << scale;
  if (flag == 1) pos = (pos/zone_size) * zone_size;
  next = pos + BLOCK_SIZE - 1;

  /* If 'pos' is in the last block of a zone, do not clear the zone. */
  if (next/zone_size != pos/zone_size) return;
  if ( (blo = read_map(rip, next)) == NO_BLOCK) return;
  bhi = (  ((blo>>scale)+1) << scale)   - 1;

  /* Clear all the blocks between 'blo' and 'bhi'. */
  for (b = blo; b <= bhi; b++) {
        bp = get_block(rip->i_dev, b, NO_READ);
        zero_block(bp);
        put_block(bp, FULL_DATA_BLOCK);
  }
}


/*===========================================================================*
 *                              new_block                                    *
 *===========================================================================*/
```

```
PUBLIC struct buf *new_block(rip, position)
register struct inode *rip;	/* pointer to inode */
off_t position;			/* file pointer */
{
/* Acquire a new block and return a pointer to it.  Doing so may require
 * allocating a complete zone, and then returning the initial block.
 * On the other hand, the current zone may still have some unused blocks.
 */

  register struct buf *bp;
  block_t b, base_block;
  zone_t z;
  zone_t zone_size;
  int scale, r;
  struct super_block *sp;

  /* Is another block available in the current zone? */
  if ( (b = read_map(rip, position)) == NO_BLOCK) {
	/* Choose first zone if possible. */
	/* Lose if the file is nonempty but the first zone number is NO_ZONE
	 * corresponding to a zone full of zeros.  It would be better to
	 * search near the last real zone.
	 */
	if (rip->i_zone[0] == NO_ZONE) {
		sp = rip->i_sp;
		z = sp->s_firstdatazone;
	} else {
		z = rip->i_zone[0];	/* hunt near first zone */
	}
	if ( (z = alloc_zone(rip->i_dev, z)) == NO_ZONE) return(NIL_BUF);
	if ( (r = write_map(rip, position, z)) != OK) {
		free_zone(rip->i_dev, z);
		err_code = r;
		return(NIL_BUF);
	}

	/* If we are not writing at EOF, clear the zone, just to be safe. */
	if ( position != rip->i_size) clear_zone(rip, position, 1);
	scale = rip->i_sp->s_log_zone_size;
	base_block = (block_t) z << scale;
	zone_size = (zone_t) BLOCK_SIZE << scale;
	b = base_block + (block_t)((position % zone_size)/BLOCK_SIZE);
  }

  bp = get_block(rip->i_dev, b, NO_READ);
  zero_block(bp);
  return(bp);
}


/*===========================================================================*
 *				zero_block				     *
 *===========================================================================*/
PUBLIC void zero_block(bp)
register struct buf *bp;	/* pointer to buffer to zero */
{
/* Zero a block. */

  memset(bp->b_data, 0, BLOCK_SIZE);
  bp->b_dirt = DIRTY;
```

```
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/pipe.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file deals with the suspension and revival of processes.  A process can
 * be suspended because it wants to read or write from a pipe and can't, or
 * because it wants to read or write from a special file and can't.  When a
 * process can't continue it is suspended, and revived later when it is able
 * to continue.
 *
 * The entry points into this file are
 *   do_pipe:     perform the PIPE system call
 *   pipe_check:  check to see that a read or write on a pipe is feasible now
 *   suspend:     suspend a process that cannot do a requested read or write
 *   release:     check to see if a suspended process can be released and do it
 *   revive:      mark a suspended process as able to run again
 *   do_unpause:  a signal has been sent to a process; see if it suspended
 */

#include "fs.h"
#include <fcntl.h>
#include <signal.h>
#include <minix/boot.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

PRIVATE message mess;

/*===========================================================================*
 *                              do_pipe                                      *
 *===========================================================================*/
PUBLIC int do_pipe()
{
/* Perform the pipe(fil_des) system call. */

  register struct fproc *rfp;
  register struct inode *rip;
  int r;
  struct filp *fil_ptr0, *fil_ptr1;
  int fil_des[2];               /* reply goes here */

  /* Acquire two file descriptors. */
  rfp = fp;
  if ( (r = get_fd(0, R_BIT, &fil_des[0], &fil_ptr0)) != OK) return(r);
  rfp->fp_filp[fil_des[0]] = fil_ptr0;
  fil_ptr0->filp_count = 1;
  if ( (r = get_fd(0, W_BIT, &fil_des[1], &fil_ptr1)) != OK) {
        rfp->fp_filp[fil_des[0]] = NIL_FILP;
        fil_ptr0->filp_count = 0;
```

```
        return(r);
  }
  rfp->fp_filp[fil_des[1]] = fil_ptr1;
  fil_ptr1->filp_count = 1;

  /* Make the inode on the pipe device. */
  if ( (rip = alloc_inode(PIPE_DEV, I_REGULAR) ) == NIL_INODE) {
        rfp->fp_filp[fil_des[0]] = NIL_FILP;
        fil_ptr0->filp_count = 0;
        rfp->fp_filp[fil_des[1]] = NIL_FILP;
        fil_ptr1->filp_count = 0;
        return(err_code);
  }

  if (read_only(rip) != OK) panic("pipe device is read only", NO_NUM);

  rip->i_pipe = I_PIPE;
  rip->i_mode &= ~I_REGULAR;
  rip->i_mode |= I_NAMED_PIPE;  /* pipes and FIFOs have this bit set */
  fil_ptr0->filp_ino = rip;
  fil_ptr0->filp_flags = O_RDONLY;
  dup_inode(rip);               /* for double usage */
  fil_ptr1->filp_ino = rip;
  fil_ptr1->filp_flags = O_WRONLY;
  rw_inode(rip, WRITING);       /* mark inode as allocated */
  reply_i1 = fil_des[0];
  reply_i2 = fil_des[1];
  rip->i_update = ATIME | CTIME | MTIME;
  return(OK);
}


/*===========================================================================*
 *                              pipe_check                                   *
 *===========================================================================*/
PUBLIC int pipe_check(rip, rw_flag, oflags, bytes, position, canwrite)
register struct inode *rip;     /* the inode of the pipe */
int rw_flag;                    /* READING or WRITING */
int oflags;                     /* flags set by open or fcntl */
register int bytes;             /* bytes to be read or written (all chunks) */
register off_t position;        /* current file position */
int *canwrite;                  /* return: number of bytes we can write */
{
/* Pipes are a little different.  If a process reads from an empty pipe for
 * which a writer still exists, suspend the reader.  If the pipe is empty
 * and there is no writer, return 0 bytes.  If a process is writing to a
 * pipe and no one is reading from it, give a broken pipe error.
 */

  int r = 0;

  /* If reading, check for empty pipe. */
  if (rw_flag == READING) {
        if (position >= rip->i_size) {
                /* Process is reading from an empty pipe. */
                if (find_filp(rip, W_BIT) != NIL_FILP) {
                        /* Writer exists */
                        if (oflags & O_NONBLOCK)
                                r = EAGAIN;
                        else
```

```
                                suspend(XPIPE); /* block reader */

                        /* If need be, activate sleeping writers. */
                        if (susp_count > 0) release(rip, WRITE, susp_count);
                }
                return(r);
        }
  } else {
        /* Process is writing to a pipe. */
/*      if (bytes > PIPE_SIZE) return(EFBIG); */
        if (find_filp(rip, R_BIT) == NIL_FILP) {
                /* Tell kernel to generate a SIGPIPE signal. */
                sys_kill((int)(fp - fproc), SIGPIPE);
                return(EPIPE);
        }

        if (position + bytes > PIPE_SIZE) {
                if ((oflags & O_NONBLOCK) && bytes < PIPE_SIZE)
                        return(EAGAIN);
                else if ((oflags & O_NONBLOCK) && bytes > PIPE_SIZE) {
                        if ( (*canwrite = (PIPE_SIZE - position)) > 0)  {
                                /* Do a partial write. Need to wakeup reader */
                                release(rip, READ, susp_count);
                                return(1);
                        } else {
                                return(EAGAIN);
                        }
                     }
                if (bytes > PIPE_SIZE) {
                        if ((*canwrite = PIPE_SIZE - position) > 0) {
                                /* Do a partial write. Need to wakeup reader
                                 * since we'll suspend ourself in read_write()
                                 */
                                release(rip, READ, susp_count);
                                return(1);
                        }
                }
                suspend(XPIPE); /* stop writer -- pipe full */
                return(0);
        }

        /* Writing to an empty pipe.  Search for suspended reader. */
        if (position == 0) release(rip, READ, susp_count);
  }

  *canwrite = 0;
  return(1);
}


/*===========================================================================*
 *                              suspend                                      *
 *===========================================================================*/
PUBLIC void suspend(task)
int task;                       /* who is proc waiting for? (PIPE = pipe) */
{
/* Take measures to suspend the processing of the present system call.
 * Store the parameters to be used upon resuming in the process table.
 * (Actually they are not used when a process is waiting for an I/O device,
 * but they are needed for pipes, and it is not worth making the distinction.)
```

```
 */

  if (task == XPIPE || task == XPOPEN) susp_count++;/* #procs susp'ed on pipe*/
  fp->fp_suspended = SUSPENDED;
  fp->fp_fd = fd << 8 | fs_call;
  fp->fp_task = -task;
  if (task == XLOCK) {
        fp->fp_buffer = (char *) name1; /*  third arg to fcntl() */
        fp->fp_nbytes =request;         /* second arg to fcntl() */
  } else {
        fp->fp_buffer = buffer;         /* for reads and writes */
        fp->fp_nbytes = nbytes;
  }
  dont_reply = TRUE;            /* do not send caller a reply message now */
}


/*===========================================================================*
 *                              release                                      *
 *===========================================================================*/
PUBLIC void release(ip, call_nr, count)
register struct inode *ip;      /* inode of pipe */
int call_nr;                    /* READ, WRITE, OPEN or CREAT */
int count;                      /* max number of processes to release */
{
/* Check to see if any process is hanging on the pipe whose inode is in 'ip'.
 * If one is, and it was trying to perform the call indicated by 'call_nr',
 * release it.
 */

  register struct fproc *rp;

  /* Search the proc table. */
  for (rp = &fproc[0]; rp < &fproc[NR_PROCS]; rp++) {
        if (rp->fp_suspended == SUSPENDED &&
                        rp->fp_revived == NOT_REVIVING &&
                        (rp->fp_fd & BYTE) == call_nr &&
                        rp->fp_filp[rp->fp_fd>>8]->filp_ino == ip) {
                revive((int)(rp - fproc), 0);
                susp_count--;   /* keep track of who is suspended */
                if (--count == 0) return;
        }
  }
}


/*===========================================================================*
 *                              revive                                       *
 *===========================================================================*/
PUBLIC void revive(proc_nr, bytes)
int proc_nr;                    /* process to revive */
int bytes;                      /* if hanging on task, how many bytes read */
{
/* Revive a previously blocked process. When a process hangs on tty, this
 * is the way it is eventually released.
 */

  register struct fproc *rfp;
  register int task;

```

```
  if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("revive err", proc_nr);
  rfp = &fproc[proc_nr];
  if (rfp->fp_suspended == NOT_SUSPENDED || rfp->fp_revived == REVIVING)return;

  /* The 'reviving' flag only applies to pipes.  Processes waiting for TTY get
   * a message right away.  The revival process is different for TTY and pipes.
   * For TTY revival, the work is already done, for pipes it is not: the proc
   * must be restarted so it can try again.
   */
  task = -rfp->fp_task;
  if (task == XPIPE || task == XLOCK) {
        /* Revive a process suspended on a pipe or lock. */
        rfp->fp_revived = REVIVING;
        reviving++;             /* process was waiting on pipe or lock */
  } else {
        rfp->fp_suspended = NOT_SUSPENDED;
        if (task == XPOPEN) /* process blocked in open or create */
                reply(proc_nr, rfp->fp_fd>>8);
        else {
                /* Revive a process suspended on TTY or other device. */
                rfp->fp_nbytes = bytes; /*pretend it wants only what there is*/
                reply(proc_nr, bytes);  /* unblock the process */
        }
  }
}


/*===========================================================================*
 *                              do_unpause                                   *
 *===========================================================================*/
PUBLIC int do_unpause()
{
/* A signal has been sent to a user who is paused on the file system.
 * Abort the system call with the EINTR error message.
 */

  register struct fproc *rfp;
  int proc_nr, task, fild;
  struct filp *f;
  dev_t dev;

  if (who > MM_PROC_NR) return(EPERM);
  proc_nr = pro;
  if (proc_nr < 0 || proc_nr >= NR_PROCS) panic("unpause err 1", proc_nr);
  rfp = &fproc[proc_nr];
  if (rfp->fp_suspended == NOT_SUSPENDED) return(OK);
  task = -rfp->fp_task;

  switch(task) {
        case XPIPE:             /* process trying to read or write a pipe */
                break;

        case XOPEN:             /* process trying to open a special file */
                panic ("fs/do_unpause called with XOPEN\n", NO_NUM);

        case XLOCK:             /* process trying to set a lock with FCNTL */
                break;

        case XPOPEN:            /* process trying to open a fifo */
                break;
```

```

        default:                /* process trying to do device I/O (e.g. tty)*/
                fild = (rfp->fp_fd >> 8) & BYTE;/* extract file descriptor */
                if (fild < 0 || fild >= OPEN_MAX)panic("unpause err 2",NO_NUM);
                f = rfp->fp_filp[fild];
                dev = (dev_t) f->filp_ino->i_zone[0];   /* device hung on */
                mess.TTY_LINE = (dev >> MINOR) & BYTE;
                mess.PROC_NR = proc_nr;

                /* Tell kernel R or W. Mode is from current call, not open. */
                mess.COUNT = (rfp->fp_fd & BYTE) == READ ? R_BIT : W_BIT;
                mess.m_type = CANCEL;
                fp = rfp;       /* hack - call_ctty uses fp */
                (*dmap[(dev >> MAJOR) & BYTE].dmap_rw)(task, &mess);
  }

  rfp->fp_suspended = NOT_SUSPENDED;
  reply(proc_nr, EINTR);        /* signal interrupted call */
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/path.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the procedures that look up path names in the directory
 * system and determine the inode number that goes with a given path name.
 *
 *  The entry points into this file are
 *   eat_path:   the 'main' routine of the path-to-inode conversion mechanism
 *   last_dir:   find the final directory on a given path
 *   advance:    parse one component of a path name
 *   search_dir: search a directory for a string and return its inode number
 */

#include "fs.h"
#include <string.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "super.h"

PUBLIC char dot1[2] = ".";      /* used for search_dir to bypass the access */
PUBLIC char dot2[3] = "..";     /* permissions for . and ..                 */

FORWARD _PROTOTYPE( char *get_name, (char *old_name, char string [NAME_MAX]) );

/*===========================================================================*
 *                              eat_path                                     *
 *===========================================================================*/
PUBLIC struct inode *eat_path(path)
char *path;                     /* the path name to be parsed */
{
```

```
/* Parse the path 'path' and put its inode in the inode table. If not possible,
 * return NIL_INODE as function value and an error code in 'err_code'.
 */

  register struct inode *ldip, *rip;
  char string[NAME_MAX];        /* hold 1 path component name here */

  /* First open the path down to the final directory. */
  if ( (ldip = last_dir(path, string)) == NIL_INODE)
        return(NIL_INODE);      /* we couldn't open final directory */

  /* The path consisting only of "/" is a special case, check for it. */
  if (string[0] == '\0') return(ldip);

  /* Get final component of the path. */
  rip = advance(ldip, string);
  put_inode(ldip);
  return(rip);
}


/*===========================================================================*
 *                              last_dir                                     *
 *===========================================================================*/
PUBLIC struct inode *last_dir(path, string)
char *path;                     /* the path name to be parsed */
char string[NAME_MAX];          /* the final component is returned here */
{
/* Given a path, 'path', located in the fs address space, parse it as
 * far as the last directory, fetch the inode for the last directory into
 * the inode table, and return a pointer to the inode.  In
 * addition, return the final component of the path in 'string'.
 * If the last directory can't be opened, return NIL_INODE and
 * the reason for failure in 'err_code'.
 */

  register struct inode *rip;
  register char *new_name;
  register struct inode *new_ip;

  /* Is the path absolute or relative?  Initialize 'rip' accordingly. */
  rip = (*path == '/' ? fp->fp_rootdir : fp->fp_workdir);

  /* If dir has been removed or path is empty, return ENOENT. */
  if (rip->i_nlinks == 0 || *path == '\0') {
        err_code = ENOENT;
        return(NIL_INODE);
  }

  dup_inode(rip);               /* inode will be returned with put_inode */

  /* Scan the path component by component. */
  while (TRUE) {
        /* Extract one component. */
        if ( (new_name = get_name(path, string)) == (char*) 0) {
                put_inode(rip); /* bad path in user space */
                return(NIL_INODE);
        }
        if (*new_name == '\0')
                if ( (rip->i_mode & I_TYPE) == I_DIRECTORY)
```

```
                        return(rip);    /* normal exit */
                else {
                        /* last file of path prefix is not a directory */
                        put_inode(rip);
                        err_code = ENOTDIR;
                        return(NIL_INODE);
                }

        /* There is more path.  Keep parsing. */
        new_ip = advance(rip, string);
        put_inode(rip);         /* rip either obsolete or irrelevant */
        if (new_ip == NIL_INODE) return(NIL_INODE);

        /* The call to advance() succeeded.  Fetch next component. */
        path = new_name;
        rip = new_ip;
  }
}


/*===========================================================================*
 *                              get_name                                     *
 *===========================================================================*/
PRIVATE char *get_name(old_name, string)
char *old_name;                 /* path name to parse */
char string[NAME_MAX];          /* component extracted from 'old_name' */
{
/* Given a pointer to a path name in fs space, 'old_name', copy the next
 * component to 'string' and pad with zeros.  A pointer to that part of
 * the name as yet unparsed is returned.  Roughly speaking,
 * 'get_name' = 'old_name' - 'string'.
 *
 * This routine follows the standard convention that /usr/ast, /usr//ast,
 * //usr///ast and /usr/ast/ are all equivalent.
 */

  register int c;
  register char *np, *rnp;

  np = string;                  /* 'np' points to current position */
  rnp = old_name;               /* 'rnp' points to unparsed string */
  while ( (c = *rnp) == '/') rnp++;     /* skip leading slashes */

  /* Copy the unparsed path, 'old_name', to the array, 'string'. */
  while ( rnp < &old_name[PATH_MAX]  &&  c != '/'   &&  c != '\0') {
        if (np < &string[NAME_MAX]) *np++ = c;
        c = *++rnp;             /* advance to next character */
  }

  /* To make /usr/ast/ equivalent to /usr/ast, skip trailing slashes. */
  while (c == '/' && rnp < &old_name[PATH_MAX]) c = *++rnp;

  if (np < &string[NAME_MAX]) *np = '\0';       /* Terminate string */

  if (rnp >= &old_name[PATH_MAX]) {
        err_code = ENAMETOOLONG;
        return((char *) 0);
  }
  return(rnp);
}
```

```
/*===========================================================================*
 *                              advance                                      *
 *===========================================================================*/
PUBLIC struct inode *advance(dirp, string)
struct inode *dirp;             /* inode for directory to be searched */
char string[NAME_MAX];          /* component name to look for */
{
/* Given a directory and a component of a path, look up the component in
 * the directory, find the inode, open it, and return a pointer to its inode
 * slot.  If it can't be done, return NIL_INODE.
 */

  register struct inode *rip;
  struct inode *rip2;
  register struct super_block *sp;
  int r, inumb;
  dev_t mnt_dev;
  ino_t numb;

  /* If 'string' is empty, yield same inode straight away. */
  if (string[0] == '\0') return(get_inode(dirp->i_dev, (int) dirp->i_num));

  /* Check for NIL_INODE. */
  if (dirp == NIL_INODE) return(NIL_INODE);

  /* If 'string' is not present in the directory, signal error. */
  if ( (r = search_dir(dirp, string, &numb, LOOK_UP)) != OK) {
        err_code = r;
        return(NIL_INODE);
  }

  /* Don't go beyond the current root directory, unless the string is dot2. */
  if (dirp == fp->fp_rootdir && strcmp(string, "..") == 0 && string != dot2)
                return(get_inode(dirp->i_dev, (int) dirp->i_num));

  /* The component has been found in the directory.  Get inode. */
  if ( (rip = get_inode(dirp->i_dev, (int) numb)) == NIL_INODE)
        return(NIL_INODE);

  if (rip->i_num == ROOT_INODE)
        if (dirp->i_num == ROOT_INODE) {
            if (string[1] == '.') {
                for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++){
                        if (sp->s_dev == rip->i_dev) {
                                /* Release the root inode.  Replace by the
                                 * inode mounted on.
                                 */
                                put_inode(rip);
                                mnt_dev = sp->s_imount->i_dev;
                                inumb = (int) sp->s_imount->i_num;
                                rip2 = get_inode(mnt_dev, inumb);
                                rip = advance(rip2, string);
                                put_inode(rip2);
                                break;
                        }
                }
            }
        }
```

```
  if (rip == NIL_INODE) return(NIL_INODE);

  /* See if the inode is mounted on.  If so, switch to root directory of the
   * mounted file system.  The super_block provides the linkage between the
   * inode mounted on and the root directory of the mounted file system.
   */
  while (rip != NIL_INODE && rip->i_mount == I_MOUNT) {
        /* The inode is indeed mounted on. */
        for (sp = &super_block[0]; sp < &super_block[NR_SUPERS]; sp++) {
                if (sp->s_imount == rip) {
                        /* Release the inode mounted on.  Replace by the
                         * inode of the root inode of the mounted device.
                         */
                        put_inode(rip);
                        rip = get_inode(sp->s_dev, ROOT_INODE);
                        break;
                }
        }
  }
  return(rip);          /* return pointer to inode's component */
}


/*===========================================================================*
 *                              search_dir                                   *
 *===========================================================================*/
PUBLIC int search_dir(ldir_ptr, string, numb, flag)
register struct inode *ldir_ptr;        /* ptr to inode for dir to search */
char string[NAME_MAX];          /* component to search for */
ino_t *numb;                    /* pointer to inode number */
int flag;                       /* LOOK_UP, ENTER, DELETE or IS_EMPTY */
{
/* This function searches the directory whose inode is pointed to by 'ldip':
 * if (flag == ENTER)  enter 'string' in the directory with inode # '*numb';
 * if (flag == DELETE) delete 'string' from the directory;
 * if (flag == LOOK_UP) search for 'string' and return inode # in 'numb';
 * if (flag == IS_EMPTY) return OK if only . and .. in dir else ENOTEMPTY;
 *
 *    if 'string' is dot1 or dot2, no access permissions are checked.
 */

  register struct direct *dp;
  register struct buf *bp;
  int i, r, e_hit, t, match;
  mode_t bits;
  off_t pos;
  unsigned new_slots, old_slots;
  block_t b;
  struct super_block *sp;
  int extended = 0;

  /* If 'ldir_ptr' is not a pointer to a dir inode, error. */
  if ( (ldir_ptr->i_mode & I_TYPE) != I_DIRECTORY) return(ENOTDIR);

  r = OK;

  if (flag != IS_EMPTY) {
        bits = (flag == LOOK_UP ? X_BIT : W_BIT | X_BIT);

        if (string == dot1 || string == dot2) {
```

```
                if (flag != LOOK_UP) r = read_only(ldir_ptr);
                                        /* only a writable device is required. */
        }
        else r = forbidden(ldir_ptr, bits); /* check access permissions */
  }
  if (r != OK) return(r);

  /* Step through the directory one block at a time. */
  old_slots = (unsigned) (ldir_ptr->i_size/DIR_ENTRY_SIZE);
  new_slots = 0;
  e_hit = FALSE;
  match = 0;                    /* set when a string match occurs */

  for (pos = 0; pos < ldir_ptr->i_size; pos += BLOCK_SIZE) {
        b = read_map(ldir_ptr, pos);    /* get block number */

        /* Since directories don't have holes, 'b' cannot be NO_BLOCK. */
        bp = get_block(ldir_ptr->i_dev, b, NORMAL);     /* get a dir block */

        /* Search a directory block. */
        for (dp = &bp->b_dir[0]; dp < &bp->b_dir[NR_DIR_ENTRIES]; dp++) {
                if (++new_slots > old_slots) { /* not found, but room left */
                        if (flag == ENTER) e_hit = TRUE;
                        break;
                }

                /* Match occurs if string found. */
                if (flag != ENTER && dp->d_ino != 0) {
                        if (flag == IS_EMPTY) {
                                /* If this test succeeds, dir is not empty. */
                                if (strcmp(dp->d_name, "." ) != 0 &&
                                    strcmp(dp->d_name, "..") != 0) match = 1;
                        } else {
                                if (strncmp(dp->d_name, string, NAME_MAX) == 0)
                                        match = 1;
                        }
                }

                if (match) {
                        /* LOOK_UP or DELETE found what it wanted. */
                        r = OK;
                        if (flag == IS_EMPTY) r = ENOTEMPTY;
                        else if (flag == DELETE) {
                                /* Save d_ino for recovery. */
                                t = NAME_MAX - sizeof(ino_t);
                                *((ino_t *) &dp->d_name[t]) = dp->d_ino;
                                dp->d_ino = 0;  /* erase entry */
                                bp->b_dirt = DIRTY;
                                ldir_ptr->i_update |= CTIME | MTIME;
                                ldir_ptr->i_dirt = DIRTY;
                        } else {
                                sp = ldir_ptr->i_sp;    /* 'flag' is LOOK_UP */
                                *numb = conv2(sp->s_native, (int) dp->d_ino);
                        }
                        put_block(bp, DIRECTORY_BLOCK);
                        return(r);
                }


                /* Check for free slot for the benefit of ENTER. */
```

```
                if (flag == ENTER && dp->d_ino == 0) {
                        e_hit = TRUE;   /* we found a free slot */
                        break;
                }
        }

        /* The whole block has been searched or ENTER has a free slot. */
        if (e_hit) break;       /* e_hit set if ENTER can be performed now */
        put_block(bp, DIRECTORY_BLOCK); /* otherwise, continue searching dir */
  }

  /* The whole directory has now been searched. */
  if (flag != ENTER) return(flag == IS_EMPTY ? OK : ENOENT);

  /* This call is for ENTER.  If no free slot has been found so far, try to
   * extend directory.
   */
  if (e_hit == FALSE) { /* directory is full and no room left in last block */
        new_slots++;            /* increase directory size by 1 entry */
        if (new_slots == 0) return(EFBIG); /* dir size limited by slot count */
        if ( (bp = new_block(ldir_ptr, ldir_ptr->i_size)) == NIL_BUF)
                return(err_code);
        dp = &bp->b_dir[0];
        extended = 1;
  }

  /* 'bp' now points to a directory block with space. 'dp' points to slot. */
  (void) memset(dp->d_name, 0, (size_t) NAME_MAX); /* clear entry */
  for (i = 0; string[i] && i < NAME_MAX; i++) dp->d_name[i] = string[i];
  sp = ldir_ptr->i_sp;
  dp->d_ino = conv2(sp->s_native, (int) *numb);
  bp->b_dirt = DIRTY;
  put_block(bp, DIRECTORY_BLOCK);
  ldir_ptr->i_update |= CTIME | MTIME;  /* mark mtime for update later */
  ldir_ptr->i_dirt = DIRTY;
  if (new_slots > old_slots) {
        ldir_ptr->i_size = (off_t) new_slots * DIR_ENTRY_SIZE;
        /* Send the change to disk if the directory is extended. */
        if (extended) rw_inode(ldir_ptr, WRITING);
  }
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/mount.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file performs the MOUNT and UMOUNT system calls.
 *
 * The entry points into this file are
 *   do_mount:  perform the MOUNT system call
 *   do_umount: perform the UMOUNT system call
 */

#include "fs.h"
#include <fcntl.h>
#include <minix/com.h>
```

```c
#include <sys/stat.h>
#include "buf.h"
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

PRIVATE message dev_mess;

FORWARD _PROTOTYPE( dev_t name_to_dev, (char *path)                     );

/*===========================================================================*
 *                              do_mount                                     *
 *===========================================================================*/
PUBLIC int do_mount()
{
/* Perform the mount(name, mfile, rd_only) system call. */

  register struct inode *rip, *root_ip;
  struct super_block *xp, *sp;
  dev_t dev;
  mode_t bits;
  int rdir, mdir;               /* TRUE iff {root|mount} file is dir */
  int r, found, major, task;

  /* Only the super-user may do MOUNT. */
  if (!super_user) return(EPERM);

  /* If 'name' is not for a block special file, return error. */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);

  /* Scan super block table to see if dev already mounted & find a free slot.*/
  sp = NIL_SUPER;
  found = FALSE;
  for (xp = &super_block[0]; xp < &super_block[NR_SUPERS]; xp++) {
        if (xp->s_dev == dev) found = TRUE;     /* is it mounted already? */
        if (xp->s_dev == NO_DEV) sp = xp;       /* record free slot */
  }
  if (found) return(EBUSY);     /* already mounted */
  if (sp == NIL_SUPER) return(ENFILE);  /* no super block available */

  dev_mess.m_type = DEV_OPEN;           /* distinguish from close */
  dev_mess.DEVICE = dev;                /* Touch the device. */
  if (rd_only) dev_mess.COUNT = R_BIT;
  else  dev_mess.COUNT = R_BIT|W_BIT;

  major = (dev >> MAJOR) & BYTE;
  if (major <= 0 || major >= max_major) return(ENODEV);
  task = dmap[major].dmap_task;         /* device task nr */
  (*dmap[major].dmap_open)(task, &dev_mess);
  if (dev_mess.REP_STATUS != OK) return(EINVAL);

  /* Fill in the super block. */
  sp->s_dev = dev;              /* read_super() needs to know which dev */
  r = read_super(sp);

  /* Is it recognized as a Minix filesystem? */
```

```
  if (r != OK) {
        dev_mess.m_type = DEV_CLOSE;
        dev_mess.DEVICE = dev;
        (*dmap[major].dmap_close)(task, &dev_mess);
        return(r);
  }

  /* Now get the inode of the file to be mounted on. */
  if (fetch_name(name2, name2_length, M1) != OK) {
        sp->s_dev = NO_DEV;
        dev_mess.m_type = DEV_CLOSE;
        dev_mess.DEVICE = dev;
        (*dmap[major].dmap_close)(task, &dev_mess);
        return(err_code);
  }
  if ( (rip = eat_path(user_path)) == NIL_INODE) {
        sp->s_dev = NO_DEV;
        dev_mess.m_type = DEV_CLOSE;
        dev_mess.DEVICE = dev;
        (*dmap[major].dmap_close)(task, &dev_mess);
        return(err_code);
  }

  /* It may not be busy. */
  r = OK;
  if (rip->i_count > 1) r = EBUSY;

  /* It may not be special. */
  bits = rip->i_mode & I_TYPE;
  if (bits == I_BLOCK_SPECIAL || bits == I_CHAR_SPECIAL) r = ENOTDIR;

  /* Get the root inode of the mounted file system. */
  root_ip = NIL_INODE;          /* if 'r' not OK, make sure this is defined */
  if (r == OK) {
        if ( (root_ip = get_inode(dev, ROOT_INODE)) == NIL_INODE) r = err_code;
  }
  if (root_ip != NIL_INODE && root_ip->i_mode == 0) r = EINVAL;

  /* File types of 'rip' and 'root_ip' may not conflict. */
  if (r == OK) {
        mdir = ((rip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */
        rdir = ((root_ip->i_mode & I_TYPE) == I_DIRECTORY);
        if (!mdir && rdir) r = EISDIR;
  }

  /* If error, return the super block and both inodes; release the maps. */
  if (r != OK) {
        put_inode(rip);
        put_inode(root_ip);
        (void) do_sync();
        invalidate(dev);

        sp->s_dev = NO_DEV;
        dev_mess.m_type = DEV_CLOSE;
        dev_mess.DEVICE = dev;
        (*dmap[major].dmap_close)(task, &dev_mess);
        return(r);
  }

  /* Nothing else can go wrong.  Perform the mount. */
```

```
  rip->i_mount = I_MOUNT;       /* this bit says the inode is mounted on */
  sp->s_imount = rip;
  sp->s_isup = root_ip;
  sp->s_rd_only = rd_only;
  return(OK);
}


/*===========================================================================*
 *                              do_umount                                    *
 *===========================================================================*/
PUBLIC int do_umount()
{
/* Perform the umount(name) system call. */

  register struct inode *rip;
  struct super_block *sp, *sp1;
  dev_t dev;
  int count;
  int major, task;

  /* Only the super-user may do UMOUNT. */
  if (!super_user) return(EPERM);

  /* If 'name' is not for a block special file, return error. */
  if (fetch_name(name, name_length, M3) != OK) return(err_code);
  if ( (dev = name_to_dev(user_path)) == NO_DEV) return(err_code);

  /* See if the mounted device is busy.  Only 1 inode using it should be
   * open -- the root inode -- and that inode only 1 time.
   */
  count = 0;
  for (rip = &inode[0]; rip< &inode[NR_INODES]; rip++)
        if (rip->i_count > 0 && rip->i_dev == dev) count += rip->i_count;
  if (count > 1) return(EBUSY); /* can't umount a busy file system */

  /* Find the super block. */
  sp = NIL_SUPER;
  for (sp1 = &super_block[0]; sp1 < &super_block[NR_SUPERS]; sp1++) {
        if (sp1->s_dev == dev) {
                sp = sp1;
                break;
        }
  }

  /* Sync the disk, and invalidate cache. */
  (void) do_sync();             /* force any cached blocks out of memory */
  invalidate(dev);              /* invalidate cache entries for this dev */
  if (sp == NIL_SUPER) return(EINVAL);

  major = (dev >> MAJOR) & BYTE;        /* major device nr */
  task = dmap[major].dmap_task; /* device task nr */
  dev_mess.m_type = DEV_CLOSE;          /* distinguish from open */
  dev_mess.DEVICE = dev;
  (*dmap[major].dmap_close)(task, &dev_mess);

  /* Finish off the unmount. */
  sp->s_imount->i_mount = NO_MOUNT;     /* inode returns to normal */
  put_inode(sp->s_imount);      /* release the inode mounted on */
  put_inode(sp->s_isup);        /* release the root inode of the mounted fs */
```

```
  sp->s_imount = NIL_INODE;
  sp->s_dev = NO_DEV;
  return(OK);
}


/*===========================================================================*
 *                              name_to_dev                                  *
 *===========================================================================*/
PRIVATE dev_t name_to_dev(path)
char *path;                     /* pointer to path name */
{
/* Convert the block special file 'path' to a device number.  If 'path'
 * is not a block special file, return error code in 'err_code'.
 */

  register struct inode *rip;
  register dev_t dev;

  /* If 'path' can't be opened, give up immediately. */
  if ( (rip = eat_path(path)) == NIL_INODE) return(NO_DEV);

  /* If 'path' is not a block special file, return error. */
  if ( (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) {
        err_code = ENOTBLK;
        put_inode(rip);
        return(NO_DEV);
  }

  /* Extract the device number. */
  dev = (dev_t) rip->i_zone[0];
  put_inode(rip);
  return(dev);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/link.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file handles the LINK and UNLINK system calls.  It also deals with
 * deallocating the storage used by a file when the last UNLINK is done to a
 * file and the blocks must be returned to the free block pool.
 *
 * The entry points into this file are
 *   do_link:   perform the LINK system call
 *   do_unlink: perform the UNLINK and RMDIR system calls
 *   do_rename: perform the RENAME system call
 *   truncate:  release all the blocks associated with an inode
 */

#include "fs.h"
#include <sys/stat.h>
#include <string.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
```

```
#include "super.h"

#define SAME 1000

FORWARD _PROTOTYPE( int remove_dir, (struct inode *rldirp, struct inode *rip,
                        char dir_name[NAME_MAX])                        );

FORWARD _PROTOTYPE( int unlink_file, (struct inode *dirp, struct inode *rip,
                        char file_name[NAME_MAX])                       );


/*===========================================================================*
 *                              do_link                                      *
 *===========================================================================*/
PUBLIC int do_link()
{
/* Perform the link(name1, name2) system call. */

  register struct inode *ip, *rip;
  register int r;
  char string[NAME_MAX];
  struct inode *new_ip;

  /* See if 'name' (file to be linked) exists. */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Check to see if the file has maximum number of links already. */
  r = OK;
  if ( (rip->i_nlinks & BYTE) >= LINK_MAX) r = EMLINK;

  /* Only super_user may link to directories. */
  if (r == OK)
        if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;

  /* If error with 'name', return the inode. */
  if (r != OK) {
        put_inode(rip);
        return(r);
  }

  /* Does the final directory of 'name2' exist? */
  if (fetch_name(name2, name2_length, M1) != OK) {
        put_inode(rip);
        return(err_code);
  }
  if ( (ip = last_dir(user_path, string)) == NIL_INODE) r = err_code;

  /* If 'name2' exists in full (even if no space) set 'r' to error. */
  if (r == OK) {
        if ( (new_ip = advance(ip, string)) == NIL_INODE) {
                r = err_code;
                if (r == ENOENT) r = OK;
        } else {
                put_inode(new_ip);
                r = EEXIST;
        }
  }

  /* Check for links across devices. */
```

```
  if (r == OK)
        if (rip->i_dev != ip->i_dev) r = EXDEV;

  /* Try to link. */
  if (r == OK)
        r = search_dir(ip, string, &rip->i_num, ENTER);

  /* If success, register the linking. */
  if (r == OK) {
        rip->i_nlinks++;
        rip->i_update |= CTIME;
        rip->i_dirt = DIRTY;
  }

  /* Done.  Release both inodes. */
  put_inode(rip);
  put_inode(ip);
  return(r);
}


/*===========================================================================*
 *                              do_unlink                                    *
 *===========================================================================*/
PUBLIC int do_unlink()
{
/* Perform the unlink(name) or rmdir(name) system call. The code for these two
 * is almost the same.  They differ only in some condition testing.  Unlink()
 * may be used by the superuser to do dangerous things; rmdir() may not.
 */

  register struct inode *rip;
  struct inode *rldirp;
  int r;
  char string[NAME_MAX];

  /* Get the last directory in the path. */
  if (fetch_name(name, name_length, M3) != OK) return(err_code);
  if ( (rldirp = last_dir(user_path, string)) == NIL_INODE)
        return(err_code);

  /* The last directory exists.  Does the file also exist? */
  r = OK;
  if ( (rip = advance(rldirp, string)) == NIL_INODE) r = err_code;

  /* If error, return inode. */
  if (r != OK) {
        put_inode(rldirp);
        return(r);
  }

  /* Do not remove a mount point. */
  if (rip->i_num == ROOT_INODE) {
        put_inode(rldirp);
        put_inode(rip);
        return(EBUSY);
  }

  /* Now test if the call is allowed, separately for unlink() and rmdir(). */
  if (fs_call == UNLINK) {
```

```
        /* Only the su may unlink directories, but the su can unlink any dir.*/
        if ( (rip->i_mode & I_TYPE) == I_DIRECTORY && !super_user) r = EPERM;

        /* Don't unlink a file if it is the root of a mounted file system. */
        if (rip->i_num == ROOT_INODE) r = EBUSY;

        /* Actually try to unlink the file; fails if parent is mode 0 etc. */
        if (r == OK) r = unlink_file(rldirp, rip, string);

  } else {
        r = remove_dir(rldirp, rip, string); /* call is RMDIR */
  }

  /* If unlink was possible, it has been done, otherwise it has not. */
  put_inode(rip);
  put_inode(rldirp);
  return(r);
}


/*===========================================================================*
 *                              do_rename                                    *
 *===========================================================================*/
PUBLIC int do_rename()
{
/* Perform the rename(name1, name2) system call. */

  struct inode *old_dirp, *old_ip;      /* ptrs to old dir, file inodes */
  struct inode *new_dirp, *new_ip;      /* ptrs to new dir, file inodes */
  struct inode *new_superdirp, *next_new_superdirp;
  int r = OK;                           /* error flag; initially no error */
  int odir, ndir;                       /* TRUE iff {old|new} file is dir */
  int same_pdir;                        /* TRUE iff parent dirs are the same */
  char old_name[NAME_MAX], new_name[NAME_MAX];
  ino_t numb;
  int r1;

  /* See if 'name1' (existing file) exists.  Get dir and file inodes. */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  if ( (old_dirp = last_dir(user_path, old_name))==NIL_INODE) return(err_code);

  if ( (old_ip = advance(old_dirp, old_name)) == NIL_INODE) r = err_code;

  /* See if 'name2' (new name) exists.  Get dir and file inodes. */
  if (fetch_name(name2, name2_length, M1) != OK) r = err_code;
  if ( (new_dirp = last_dir(user_path, new_name)) == NIL_INODE) r = err_code;
  new_ip = advance(new_dirp, new_name); /* not required to exist */

  if (old_ip != NIL_INODE)
        odir = ((old_ip->i_mode & I_TYPE) == I_DIRECTORY);  /* TRUE iff dir */

  /* If it is ok, check for a variety of possible errors. */
  if (r == OK) {
        same_pdir = (old_dirp == new_dirp);

        /* The old inode must not be a superdirectory of the new last dir. */
        if (odir && !same_pdir) {
                dup_inode(new_superdirp = new_dirp);
                while (TRUE) {          /* may hang in a file system loop */
                        if (new_superdirp == old_ip) {
```

```
                                r = EINVAL;
                                break;
                        }
                        next_new_superdirp = advance(new_superdirp, dot2);
                        put_inode(new_superdirp);
                        if (next_new_superdirp == new_superdirp)
                                break;  /* back at system root directory */
                        new_superdirp = next_new_superdirp;
                        if (new_superdirp == NIL_INODE) {
                                /* Missing ".." entry.  Assume the worst. */
                                r = EINVAL;
                                break;
                        }
                }
                put_inode(new_superdirp);
        }

        /* The old or new name must not be . or .. */
        if (strcmp(old_name, ".")==0 || strcmp(old_name, "..")==0 ||
            strcmp(new_name, ".")==0 || strcmp(new_name, "..")==0) r = EINVAL;

        /* Both parent directories must be on the same device. */
        if (old_dirp->i_dev != new_dirp->i_dev) r = EXDEV;

        /* Parent dirs must be writable, searchable and on a writable device */
        if ((r1 = forbidden(old_dirp, W_BIT | X_BIT)) != OK ||
            (r1 = forbidden(new_dirp, W_BIT | X_BIT)) != OK) r = r1;

        /* Some tests apply only if the new path exists. */
        if (new_ip == NIL_INODE) {
                /* don't rename a file with a file system mounted on it. */
                if (old_ip->i_dev != old_dirp->i_dev) r = EXDEV;
                if (odir && (new_dirp->i_nlinks & BYTE) >= LINK_MAX &&
                    !same_pdir && r == OK) r = EMLINK;
        } else {
                if (old_ip == new_ip) r = SAME; /* old=new */

                /* has the old file or new file a file system mounted on it? */
                if (old_ip->i_dev != new_ip->i_dev) r = EXDEV;

                ndir = ((new_ip->i_mode & I_TYPE) == I_DIRECTORY); /* dir ? */
                if (odir == TRUE && ndir == FALSE) r = ENOTDIR;
                if (odir == FALSE && ndir == TRUE) r = EISDIR;
        }
  }

  /* If a process has another root directory than the system root, we might
   * "accidently" be moving it's working directory to a place where it's
   * root directory isn't a super directory of it anymore. This can make
   * the function chroot useless. If chroot will be used often we should
   * probably check for it here.
   */

  /* The rename will probably work. Only two things can go wrong now:
   * 1. being unable to remove the new file. (when new file already exists)
   * 2. being unable to make the new directory entry. (new file doesn't exists)
   *     [directory has to grow by one block and cannot because the disk
   *      is completely full].
   */
  if (r == OK) {
```

```
        if (new_ip != NIL_INODE) {
                  /* There is already an entry for 'new'. Try to remove it. */
                if (odir)
                        r = remove_dir(new_dirp, new_ip, new_name);
                else
                        r = unlink_file(new_dirp, new_ip, new_name);
        }
        /* if r is OK, the rename will succeed, while there is now an
         * unused entry in the new parent directory.
         */
  }

  if (r == OK) {
        /* If the new name will be in the same parent directory as the old one,
         * first remove the old name to free an entry for the new name,
         * otherwise first try to create the new name entry to make sure
         * the rename will succeed.
         */
        numb = old_ip->i_num;           /* inode number of old file */

        if (same_pdir) {
                r = search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
                                                /* shouldn't go wrong. */
                if (r==OK) (void) search_dir(old_dirp, new_name, &numb, ENTER);
        } else {
                r = search_dir(new_dirp, new_name, &numb, ENTER);
                if (r == OK)
                    (void) search_dir(old_dirp, old_name, (ino_t *) 0, DELETE);
        }
  }
  /* If r is OK, the ctime and mtime of old_dirp and new_dirp have been marked
   * for update in search_dir.
   */

  if (r == OK && odir && !same_pdir) {
        /* Update the .. entry in the directory (still points to old_dirp). */
        numb = new_dirp->i_num;
        (void) unlink_file(old_ip, NIL_INODE, dot2);
        if (search_dir(old_ip, dot2, &numb, ENTER) == OK) {
                /* New link created. */
                new_dirp->i_nlinks++;
                new_dirp->i_dirt = DIRTY;
        }
  }

  /* Release the inodes. */
  put_inode(old_dirp);
  put_inode(old_ip);
  put_inode(new_dirp);
  put_inode(new_ip);
  return(r == SAME ? OK : r);
}


/*===========================================================================*
 *                              truncate                                     *
 *===========================================================================*/
PUBLIC void truncate(rip)
register struct inode *rip;     /* pointer to inode to be truncated */
{
```

```
/* Remove all the zones from the inode 'rip' and mark it dirty. */

  register block_t b;
  zone_t z, zone_size, z1;
  off_t position;
  int i, scale, file_type, waspipe, single, nr_indirects;
  struct buf *bp;
  dev_t dev;

  file_type = rip->i_mode & I_TYPE;     /* check to see if file is special */
  if (file_type == I_CHAR_SPECIAL || file_type == I_BLOCK_SPECIAL) return;
  dev = rip->i_dev;             /* device on which inode resides */
  scale = rip->i_sp->s_log_zone_size;
  zone_size = (zone_t) BLOCK_SIZE << scale;
  nr_indirects = rip->i_nindirs;

  /* Pipes can shrink, so adjust size to make sure all zones are removed. */
  waspipe = rip->i_pipe == I_PIPE;      /* TRUE is this was a pipe */
  if (waspipe) rip->i_size = PIPE_SIZE;

  /* Step through the file a zone at a time, finding and freeing the zones. */
  for (position = 0; position < rip->i_size; position += zone_size) {
        if ( (b = read_map(rip, position)) != NO_BLOCK) {
                z = (zone_t) b >> scale;
                free_zone(dev, z);
        }
  }

  /* All the data zones have been freed.  Now free the indirect zones. */
  rip->i_dirt = DIRTY;
  if (waspipe) {
        wipe_inode(rip);        /* clear out inode for pipes */
        return;                 /* indirect slots contain file positions */
  }
  single = rip->i_ndzones;
  free_zone(dev, rip->i_zone[single]);  /* single indirect zone */
  if ( (z = rip->i_zone[single+1]) != NO_ZONE) {
        /* Free all the single indirect zones pointed to by the double. */
        b = (block_t) z << scale;
        bp = get_block(dev, b, NORMAL); /* get double indirect zone */
        for (i = 0; i < nr_indirects; i++) {
                z1 = rd_indir(bp, i);
                free_zone(dev, z1);
        }

        /* Now free the double indirect zone itself. */
        put_block(bp, INDIRECT_BLOCK);
        free_zone(dev, z);
  }

  /* Leave zone numbers for de(1) to recover file after an unlink(2).  */
}


/*===========================================================================*
 *                              remove_dir                                   *
 *===========================================================================*/
PRIVATE int remove_dir(rldirp, rip, dir_name)
struct inode *rldirp;                   /* parent directory */
struct inode *rip;                      /* directory to be removed */
```

```c
char dir_name[NAME_MAX];                /* name of directory to be removed */
{
  /* A directory file has to be removed. Five conditions have to met:
   *    - The file must be a directory
   *    - The directory must be empty (except for . and ..)
   *    - The final component of the path must not be . or ..
   *    - The directory must not be the root of a mounted file system
   *    - The directory must not be anybody's root/working directory
   */

  int r;
  register struct fproc *rfp;

  /* search_dir checks that rip is a directory too. */
  if ((r = search_dir(rip, "", (ino_t *) 0, IS_EMPTY)) != OK) return r;

  if (strcmp(dir_name, ".") == 0 || strcmp(dir_name, "..") == 0)return(EINVAL);
  if (rip->i_num == ROOT_INODE) return(EBUSY); /* can't remove 'root' */

  for (rfp = &fproc[INIT_PROC_NR + 1]; rfp < &fproc[NR_PROCS]; rfp++)
        if (rfp->fp_workdir == rip || rfp->fp_rootdir == rip) return(EBUSY);
                                /* can't remove anybody's working dir */

  /* Actually try to unlink the file; fails if parent is mode 0 etc. */
  if ((r = unlink_file(rldirp, rip, dir_name)) != OK) return r;

  /* Unlink . and .. from the dir. The super user can link and unlink any dir,
   * so don't make too many assumptions about them.
   */
  (void) unlink_file(rip, NIL_INODE, dot1);
  (void) unlink_file(rip, NIL_INODE, dot2);
  return(OK);
}


/*===========================================================================*
 *                              unlink_file                                  *
 *===========================================================================*/
PRIVATE int unlink_file(dirp, rip, file_name)
struct inode *dirp;             /* parent directory of file */
struct inode *rip;              /* inode of file, may be NIL_INODE too. */
char file_name[NAME_MAX];       /* name of file to be removed */
{
/* Unlink 'file_name'; rip must be the inode of 'file_name' or NIL_INODE. */

  ino_t numb;                   /* inode number */
  int   r;

  /* If rip is not NIL_INODE, it is used to get faster access to the inode. */
  if (rip == NIL_INODE) {
        /* Search for file in directory and try to get its inode. */
        err_code = search_dir(dirp, file_name, &numb, LOOK_UP);
        if (err_code == OK) rip = get_inode(dirp->i_dev, (int) numb);
        if (err_code != OK || rip == NIL_INODE) return(err_code);
  } else {
        dup_inode(rip);         /* inode will be returned with put_inode */
  }

  r = search_dir(dirp, file_name, (ino_t *) 0, DELETE);

```

```
  if (r == OK) {
        rip->i_nlinks--;        /* entry deleted from parent's dir */
        rip->i_update |= CTIME;
        rip->i_dirt = DIRTY;
  }

  put_inode(rip);
  return(r);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/stadir.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains the code for performing four system calls relating to
 * status and directories.
 *
 * The entry points into this file are
 *   do_chdir:  perform the CHDIR system call
 *   do_chroot: perform the CHROOT system call
 *   do_stat:   perform the STAT system call
 *   do_fstat:  perform the FSTAT system call
 */

#include "fs.h"
#include <sys/stat.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

FORWARD _PROTOTYPE( int change, (struct inode **iip, char *name_ptr, int len));
FORWARD _PROTOTYPE( int stat_inode, (struct inode *rip, struct filp *fil_ptr,
                        char *user_addr)                                );

/*===========================================================================*
 *                              do_chdir                                     *
 *===========================================================================*/
PUBLIC int do_chdir()
{
/* Change directory.  This function is  also called by MM to simulate a chdir
 * in order to do EXEC, etc.  It also changes the root directory, the uids and
 * gids, and the umask.
 */

  int r;
  register struct fproc *rfp;

  if (who == MM_PROC_NR) {
        rfp = &fproc[slot1];
        put_inode(fp->fp_rootdir);
        dup_inode(fp->fp_rootdir = rfp->fp_rootdir);
        put_inode(fp->fp_workdir);
        dup_inode(fp->fp_workdir = rfp->fp_workdir);

        /* MM uses access() to check permissions.  To make this work, pretend
         * that the user's real ids are the same as the user's effective ids.
         * FS calls other than access() do not use the real ids, so are not
         * affected.
```

```
         */
        fp->fp_realuid =
        fp->fp_effuid = rfp->fp_effuid;
        fp->fp_realgid =
        fp->fp_effgid = rfp->fp_effgid;
        fp->fp_umask = rfp->fp_umask;
        return(OK);
  }

  /* Perform the chdir(name) system call. */
  r = change(&fp->fp_workdir, name, name_length);
  return(r);
}


/*===========================================================================*
 *                              do_chroot                                    *
 *===========================================================================*/
PUBLIC int do_chroot()
{
/* Perform the chroot(name) system call. */

  register int r;

  if (!super_user) return(EPERM);       /* only su may chroot() */
  r = change(&fp->fp_rootdir, name, name_length);
  return(r);
}


/*===========================================================================*
 *                              change                                       *
 *===========================================================================*/
PRIVATE int change(iip, name_ptr, len)
struct inode **iip;             /* pointer to the inode pointer for the dir */
char *name_ptr;                 /* pointer to the directory name to change to */
int len;                        /* length of the directory name string */
{
/* Do the actual work for chdir() and chroot(). */

  struct inode *rip;
  register int r;

  /* Try to open the new directory. */
  if (fetch_name(name_ptr, len, M3) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* It must be a directory and also be searchable. */
  if ( (rip->i_mode & I_TYPE) != I_DIRECTORY)
        r = ENOTDIR;
  else
        r = forbidden(rip, X_BIT);      /* check if dir is searchable */

  /* If error, return inode. */
  if (r != OK) {
        put_inode(rip);
        return(r);
  }

  /* Everything is OK.  Make the change. */
```

```
  put_inode(*iip);              /* release the old directory */
  *iip = rip;                   /* acquire the new one */
  return(OK);
}


/*===========================================================================*
 *                             do_stat                                       *
 *===========================================================================*/
PUBLIC int do_stat()
{
/* Perform the stat(name, buf) system call. */

  register struct inode *rip;
  register int r;

  /* Both stat() and fstat() use the same routine to do the real work.  That
   * routine expects an inode, so acquire it temporarily.
   */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
  r = stat_inode(rip, NIL_FILP, name2); /* actually do the work.*/
  put_inode(rip);               /* release the inode */
  return(r);
}


/*===========================================================================*
 *                             do_fstat                                      *
 *===========================================================================*/
PUBLIC int do_fstat()
{
/* Perform the fstat(fd, buf) system call. */

  register struct filp *rfilp;

  /* Is the file descriptor valid? */
  if ( (rfilp = get_filp(fd)) == NIL_FILP) return(err_code);

  return(stat_inode(rfilp->filp_ino, rfilp, buffer));
}


/*===========================================================================*
 *                             stat_inode                                    *
 *===========================================================================*/
PRIVATE int stat_inode(rip, fil_ptr, user_addr)
register struct inode *rip;     /* pointer to inode to stat */
struct filp *fil_ptr;           /* filp pointer, supplied by 'fstat' */
char *user_addr;                /* user space address where stat buf goes */
{
/* Common code for stat and fstat system calls. */

  struct stat statbuf;
  mode_t mo;
  int r, s;

  /* Update the atime, ctime, and mtime fields in the inode, if need be. */
  if (rip->i_update) update_times(rip);

```

```
  /* Fill in the statbuf struct. */
  mo = rip->i_mode & I_TYPE;
  s = (mo == I_CHAR_SPECIAL || mo == I_BLOCK_SPECIAL);  /* true iff special */
  statbuf.st_dev = rip->i_dev;
  statbuf.st_ino = rip->i_num;
  statbuf.st_mode = rip->i_mode;
  statbuf.st_nlink = rip->i_nlinks & BYTE;
  statbuf.st_uid = rip->i_uid;
  statbuf.st_gid = rip->i_gid & BYTE;
  statbuf.st_rdev = (dev_t) (s ? rip->i_zone[0] : NO_DEV);
  statbuf.st_size = rip->i_size;

  if (rip->i_pipe == I_PIPE) {
        statbuf.st_mode &= ~I_REGULAR;  /* wipe out I_REGULAR bit for pipes */
        if (fil_ptr != NIL_FILP && fil_ptr->filp_mode & R_BIT)
                statbuf.st_size -= fil_ptr->filp_pos;
  }

  statbuf.st_atime = rip->i_atime;
  statbuf.st_mtime = rip->i_mtime;
  statbuf.st_ctime = rip->i_ctime;

  /* Copy the struct to user space. */
  r = sys_copy(FS_PROC_NR, D, (phys_bytes) &statbuf,
                who, D, (phys_bytes) user_addr, (phys_bytes) sizeof(statbuf));
  return(r);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/protect.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file deals with protection in the file system.  It contains the code
 * for four system calls that relate to protection.
 *
 * The entry points into this file are
 *   do_chmod:  perform the CHMOD system call
 *   do_chown:  perform the CHOWN system call
 *   do_umask:  perform the UMASK system call
 *   do_access: perform the ACCESS system call
 *   forbidden: check to see if a given access is allowed on a given inode
 */

#include "fs.h"
#include <unistd.h>
#include <minix/callnr.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"
#include "super.h"

/*===========================================================================*
 *                              do_chmod                                     *
 *===========================================================================*/
PUBLIC int do_chmod()
```

```
{
/* Perform the chmod(name, mode) system call. */

  register struct inode *rip;
  register int r;

  /* Temporarily open the file. */
  if (fetch_name(name, name_length, M3) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Only the owner or the super_user may change the mode of a file.
   * No one may change the mode of a file on a read-only file system.
   */
  if (rip->i_uid != fp->fp_effuid && !super_user)
        r = EPERM;
  else
        r = read_only(rip);

  /* If error, return inode. */
  if (r != OK)  {
        put_inode(rip);
        return(r);
  }

  /* Now make the change. Clear setgid bit if file is not in caller's grp */
  rip->i_mode = (rip->i_mode & ~ALL_MODES) | (mode & ALL_MODES);
  if (!super_user && rip->i_gid != fp->fp_effgid)rip->i_mode &= ~I_SET_GID_BIT;
  rip->i_update |= CTIME;
  rip->i_dirt = DIRTY;

  put_inode(rip);
  return(OK);
}


/*===========================================================================*
 *                              do_chown                                     *
 *===========================================================================*/
PUBLIC int do_chown()
{
/* Perform the chown(name, owner, group) system call. */

  register struct inode *rip;
  register int r;

  /* Temporarily open the file. */
  if (fetch_name(name1, name1_length, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Not permitted to change the owner of a file on a read-only file sys. */
  r = read_only(rip);
  if (r == OK) {
        /* FS is R/W.  Whether call is allowed depends on ownership, etc. */
        if (super_user) {
                /* The super user can do anything. */
                rip->i_uid = owner;     /* others later */
        } else {
                /* Regular users can only change groups of their own files. */
                if (rip->i_uid != fp->fp_effuid) r = EPERM;
                if (rip->i_uid != owner) r = EPERM;     /* no giving away */
```

```
                if (fp->fp_effgid != group) r = EPERM;
        }
  }
  if (r == OK) {
        rip->i_gid = group;
        rip->i_mode &= ~(I_SET_UID_BIT | I_SET_GID_BIT);
        rip->i_update |= CTIME;
        rip->i_dirt = DIRTY;
  }

  put_inode(rip);
  return(r);
}


/*===========================================================================*
 *                              do_umask                                     *
 *===========================================================================*/
PUBLIC int do_umask()
{
/* Perform the umask(co_mode) system call. */
  register mode_t r;

  r = ~fp->fp_umask;            /* set 'r' to complement of old mask */
  fp->fp_umask = ~(co_mode & RWX_MODES);
  return(r);                    /* return complement of old mask */
}


/*===========================================================================*
 *                              do_access                                    *
 *===========================================================================*/
PUBLIC int do_access()
{
/* Perform the access(name, mode) system call. */

  struct inode *rip;
  register int r;

  /* First check to see if the mode is correct. */
  if ( (mode & ~(R_OK | W_OK | X_OK)) != 0 && mode != F_OK)
        return(EINVAL);

  /* Temporarily open the file whose access is to be checked. */
  if (fetch_name(name, name_length, M3) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Now check the permissions. */
  r = forbidden(rip, (mode_t) mode);
  put_inode(rip);
  return(r);
}


/*===========================================================================*
 *                              forbidden                                    *
 *===========================================================================*/
PUBLIC int forbidden(rip, access_desired)
register struct inode *rip;     /* pointer to inode to be checked */
mode_t access_desired;  /* RWX bits */
```

```
{
/* Given a pointer to an inode, 'rip', and the access desired, determine
 * if the access is allowed, and if not why not.  The routine looks up the
 * caller's uid in the 'fproc' table.  If access is allowed, OK is returned
 * if it is forbidden, EACCES is returned.
 */

  register struct inode *old_rip = rip;
  register struct super_block *sp;
  register mode_t bits, perm_bits;
  int r, shift, test_uid, test_gid;

  if (rip->i_mount == I_MOUNT)  /* The inode is mounted on. */
        for (sp = &super_block[1]; sp < &super_block[NR_SUPERS]; sp++)
                if (sp->s_imount == rip) {
                        rip = get_inode(sp->s_dev, ROOT_INODE);
                        break;
                } /* if */

  /* Isolate the relevant rwx bits from the mode. */
  bits = rip->i_mode;
  test_uid = (fs_call == ACCESS ? fp->fp_realuid : fp->fp_effuid);
  test_gid = (fs_call == ACCESS ? fp->fp_realgid : fp->fp_effgid);
  if (test_uid == SU_UID) {
        /* Grant read and write permission.  Grant search permission for
         * directories.  Grant execute permission (for non-directories) if
         * and only if one of the 'X' bits is set.
         */
        if ( (bits & I_TYPE) == I_DIRECTORY ||
             bits & ((X_BIT << 6) | (X_BIT << 3) | X_BIT))
                perm_bits = R_BIT | W_BIT | X_BIT;
        else
                perm_bits = R_BIT | W_BIT;
  } else {
        if (test_uid == rip->i_uid) shift = 6;          /* owner */
        else if (test_gid == rip->i_gid ) shift = 3;    /* group */
        else shift = 0;                                 /* other */
        perm_bits = (bits >> shift) & (R_BIT | W_BIT | X_BIT);
  }

  /* If access desired is not a subset of what is allowed, it is refused. */
  r = OK;
  if ((perm_bits | access_desired) != perm_bits) r = EACCES;

  /* Check to see if someone is trying to write on a file system that is
   * mounted read-only.
   */
  if (r == OK)
        if (access_desired & W_BIT) r = read_only(rip);

  if (rip != old_rip) put_inode(rip);

  return(r);
}


/*===========================================================================*
 *                              read_only                                    *
 *===========================================================================*/
PUBLIC int read_only(ip)
```

```
struct inode *ip;               /* ptr to inode whose file sys is to be cked */
{
/* Check to see if the file system on which the inode 'ip' resides is mounted
 * read only.  If so, return EROFS, else return OK.
 */

  register struct super_block *sp;

  sp = ip->i_sp;
  return(sp->s_rd_only ? EROFS : OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/time.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file takes care of those system calls that deal with time.
 *
 * The entry points into this file are
 *   do_utime:  perform the UTIME system call
 *   do_time:   perform the TIME system call
 *   do_stime:  perform the STIME system call
 *   do_tims:   perform the TIMES system call
 */

#include "fs.h"
#include <minix/callnr.h>
#include <minix/com.h>
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

PRIVATE message clock_mess;

/*===========================================================================*
 *                              do_utime                                     *
 *===========================================================================*/
PUBLIC int do_utime()
{
/* Perform the utime(name, timep) system call. */

  register struct inode *rip;
  register int len, r;

  /* Adjust for case of NULL 'timep'. */
  len = utime_length;
  if (len == 0) len = m.m2_i2;

  /* Temporarily open the file. */
  if (fetch_name(utime_file, len, M1) != OK) return(err_code);
  if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);

  /* Only the owner of a file or the super_user can change its time. */
  r = OK;
  if (rip->i_uid != fp->fp_effuid && !super_user) r = EPERM;
```

```
  if (utime_length == 0 && r != OK) r = forbidden(rip, W_BIT);
  if (read_only(rip) != OK) r = EROFS;  /* not even su can touch if R/O */
  if (r == OK) {
        if (utime_length == 0) {
                rip->i_atime = clock_time();
                rip->i_mtime = rip->i_atime;
        } else {
                rip->i_atime = utime_actime;
                rip->i_mtime = utime_modtime;
        }
        rip->i_update = CTIME;  /* discard any stale ATIME and MTIME flags */
        rip->i_dirt = DIRTY;
  }

  put_inode(rip);
  return(r);
}


/*===========================================================================*
 *                              do_time                                      *
 *===========================================================================*/
PUBLIC int do_time()

{
/* Perform the time(tp) system call. */

  reply_l1 = clock_time();      /* return time in seconds */
  return(OK);
}


/*===========================================================================*
 *                              do_stime                                     *
 *===========================================================================*/
PUBLIC int do_stime()
{
/* Perform the stime(tp) system call. */

  register int k;

  if (!super_user) return(EPERM);
  clock_mess.m_type = SET_TIME;
  clock_mess.NEW_TIME = (long) tp;
  if ( (k = sendrec(CLOCK, &clock_mess)) != OK) panic("do_stime error", k);
  return(OK);
}


/*===========================================================================*
 *                              do_tims                                      *
 *===========================================================================*/
PUBLIC int do_tims()
{
/* Perform the times(buffer) system call. */

  clock_t t[5];

  sys_times(who, t);
  reply_t1 = t[0];
```

```
  reply_t2 = t[1];
  reply_t3 = t[2];
  reply_t4 = t[3];
  reply_t5 = t[4];
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/misc.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* This file contains a collection of miscellaneous procedures.  Some of them
 * perform simple system calls.  Some others do a little part of system calls
 * that are mostly performed by the Memory Manager.
 *
 * The entry points into this file are
 *   do_dup:      perform the DUP system call
 *   do_fcntl:    perform the FCNTL system call
 *   do_sync:     perform the SYNC system call
 *   do_fork:     adjust the tables after MM has performed a FORK system call
 *   do_exec:     handle files with FD_CLOEXEC on after MM has done an EXEC
 *   do_exit:     a process has exited; note that in the tables
 *   do_set:      set uid or gid for some process
 *   do_revive:   revive a process that was waiting for something (e.g. TTY)
 */

#include "fs.h"
#include <fcntl.h>
#include <unistd.h>     /* cc runs out of memory with unistd.h :-( */
#include <minix/callnr.h>
#include <minix/com.h>
#include <minix/boot.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "dev.h"
#include "param.h"


/*===========================================================================*
 *                              do_dup                                       *
 *===========================================================================*/
PUBLIC int do_dup()
{
/* Perform the dup(fd) or dup2(fd,fd2) system call. These system calls are
 * obsolete.  In fact, it is not even possible to invoke them using the
 * current library because the library routines call fcntl().  They are
 * provided to permit old binary programs to continue to run.
 */

  register int rfd;
  register struct filp *f;
  struct filp *dummy;
  int r;

```

```
  /* Is the file descriptor valid? */
  rfd = fd & ~DUP_MASK;          /* kill off dup2 bit, if on */
  if ((f = get_filp(rfd)) == NIL_FILP) return(err_code);

  /* Distinguish between dup and dup2. */
  if (fd == rfd) {                        /* bit not on */
        /* dup(fd) */
        if ( (r = get_fd(0, 0, &fd2, &dummy)) != OK) return(r);
  } else {
        /* dup2(fd, fd2) */
        if (fd2 < 0 || fd2 >= OPEN_MAX) return(EBADF);
        if (rfd == fd2) return(fd2);    /* ignore the call: dup2(x, x) */
        fd = fd2;               /* prepare to close fd2 */
        (void) do_close();      /* cannot fail */
  }

  /* Success. Set up new file descriptors. */
  f->filp_count++;
  fp->fp_filp[fd2] = f;
  return(fd2);
}

/*===========================================================================*
 *                              do_fcntl                                     *
 *===========================================================================*/
PUBLIC int do_fcntl()
{
/* Perform the fcntl(fd, request, ...) system call. */

  register struct filp *f;
  int new_fd, r, fl;
  long cloexec_mask;            /* bit map for the FD_CLOEXEC flag */
  long clo_value;               /* FD_CLOEXEC flag in proper position */
  struct filp *dummy;

  /* Is the file descriptor valid? */
  if ((f = get_filp(fd)) == NIL_FILP) return(err_code);

  switch (request) {
     case F_DUPFD:
        /* This replaces the old dup() system call. */
        if (addr < 0 || addr >= OPEN_MAX) return(EINVAL);
        if ((r = get_fd(addr, 0, &new_fd, &dummy)) != OK) return(r);
        f->filp_count++;
        fp->fp_filp[new_fd] = f;
        return(new_fd);

     case F_GETFD:
        /* Get close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
        return( ((fp->fp_cloexec >> fd) & 01) ? FD_CLOEXEC : 0);

     case F_SETFD:
        /* Set close-on-exec flag (FD_CLOEXEC in POSIX Table 6-2). */
        cloexec_mask = 1L << fd;        /* singleton set position ok */
        clo_value = (addr & FD_CLOEXEC ? cloexec_mask : 0L);
        fp->fp_cloexec = (fp->fp_cloexec & ~cloexec_mask) | clo_value;
        return(OK);

     case F_GETFL:
        /* Get file status flags (O_NONBLOCK and O_APPEND). */
```

```
        fl = f->filp_flags & (O_NONBLOCK | O_APPEND | O_ACCMODE);
        return(fl);

     case F_SETFL:
        /* Set file status flags (O_NONBLOCK and O_APPEND). */
        fl = O_NONBLOCK | O_APPEND;
        f->filp_flags = (f->filp_flags & ~fl) | (addr & fl);
        return(OK);

     case F_GETLK:
     case F_SETLK:
     case F_SETLKW:
        /* Set or clear a file lock. */
        r = lock_op(f, request);
        return(r);

     default:
        return(EINVAL);
  }
}


/*===========================================================================*
 *                              do_sync                                      *
 *===========================================================================*/
PUBLIC int do_sync()
{
/* Perform the sync() system call.  Flush all the tables. */

  register struct inode *rip;
  register struct buf *bp;

  /* The order in which the various tables are flushed is critical.  The
   * blocks must be flushed last, since rw_inode() leaves its results in
   * the block cache.
   */

  /* Write all the dirty inodes to the disk. */
  for (rip = &inode[0]; rip < &inode[NR_INODES]; rip++)
        if (rip->i_count > 0 && rip->i_dirt == DIRTY) rw_inode(rip, WRITING);

  /* Write all the dirty blocks to the disk, one drive at a time. */
  for (bp = &buf[0]; bp < &buf[NR_BUFS]; bp++)
        if (bp->b_dev != NO_DEV && bp->b_dirt == DIRTY) flushall(bp->b_dev);

  return(OK);           /* sync() can't fail */
}


/*===========================================================================*
 *                              do_fork                                      *
 *===========================================================================*/
PUBLIC int do_fork()
{
/* Perform those aspects of the fork() system call that relate to files.
 * In particular, let the child inherit its parent's file descriptors.
 * The parent and child parameters tell who forked off whom. The file
 * system uses the same slot numbers as the kernel.  Only MM makes this call.
 */

```

```
  register struct fproc *cp;
  int i;

  /* Only MM may make this call directly. */
  if (who != MM_PROC_NR) return(EGENERIC);

  /* Copy the parent's fproc struct to the child. */
  fproc[child] = fproc[parent];

  /* Increase the counters in the 'filp' table. */
  cp = &fproc[child];
  for (i = 0; i < OPEN_MAX; i++)
        if (cp->fp_filp[i] != NIL_FILP) cp->fp_filp[i]->filp_count++;

  /* Fill in new process id. */
  cp->fp_pid = pid;

  /* A child is not a process leader. */
  cp->fp_sesldr = 0;

  /* Record the fact that both root and working dir have another user. */
  dup_inode(cp->fp_rootdir);
  dup_inode(cp->fp_workdir);
  return(OK);
}


/*===========================================================================*
 *                              do_exec                                      *
 *===========================================================================*/
PUBLIC int do_exec()
{
/* Files can be marked with the FD_CLOEXEC bit (in fp->fp_cloexec).  When
 * MM does an EXEC, it calls FS to allow FS to find these files and close them.
 */

  register int i;
  long bitmap;

  /* Only MM may make this call directly. */
  if (who != MM_PROC_NR) return(EGENERIC);

  /* The array of FD_CLOEXEC bits is in the fp_cloexec bit map. */
  fp = &fproc[slot1];           /* get_filp() needs 'fp' */
  bitmap = fp->fp_cloexec;
  if (bitmap == 0) return(OK);  /* normal case, no FD_CLOEXECs */

  /* Check the file desriptors one by one for presence of FD_CLOEXEC. */
  for (i = 0; i < OPEN_MAX; i++) {
        fd = i;
        if ( (bitmap >> i) & 01) (void) do_close();
  }

  return(OK);
}


/*===========================================================================*
 *                              do_exit                                      *
 *===========================================================================*/
```

```
PUBLIC int do_exit()
{
/* Perform the file system portion of the exit(status) system call. */

  register int i, exitee, task;
  register struct fproc *rfp;
  register struct filp *rfilp;
  register struct inode *rip;
  int major;
  dev_t dev;
  message dev_mess;

  /* Only MM may do the EXIT call directly. */
  if (who != MM_PROC_NR) return(EGENERIC);

  /* Nevertheless, pretend that the call came from the user. */
  fp = &fproc[slot1];           /* get_filp() needs 'fp' */
  exitee = slot1;

  if (fp->fp_suspended == SUSPENDED) {
        task = -fp->fp_task;
        if (task == XPIPE || task == XPOPEN) susp_count--;
        pro = exitee;
        (void) do_unpause();    /* this always succeeds for MM */
        fp->fp_suspended = NOT_SUSPENDED;
  }

  /* Loop on file descriptors, closing any that are open. */
  for (i = 0; i < OPEN_MAX; i++) {
        fd = i;
        (void) do_close();
  }

  /* Release root and working directories. */
  put_inode(fp->fp_rootdir);
  put_inode(fp->fp_workdir);
  fp->fp_rootdir = NIL_INODE;
  fp->fp_workdir = NIL_INODE;

  /* If a session leader exits then revoke access to its controlling tty from
   * all other processes using it.
   */
  if (!fp->fp_sesldr) return(OK);               /* not a session leader */
  fp->fp_sesldr = FALSE;
  if (fp->fp_tty == 0) return(OK);              /* no controlling tty */
  dev = fp->fp_tty;

  for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
        if (rfp->fp_tty == dev) rfp->fp_tty = 0;

        for (i = 0; i < OPEN_MAX; i++) {
                if ((rfilp = rfp->fp_filp[i]) == NIL_FILP) continue;
                if (rfilp->filp_mode == FILP_CLOSED) continue;
                rip = rfilp->filp_ino;
                if ((rip->i_mode & I_TYPE) != I_CHAR_SPECIAL) continue;
                if ((dev_t) rip->i_zone[0] != dev) continue;
                dev_mess.m_type = DEV_CLOSE;
                dev_mess.DEVICE = dev;
                major = (dev >> MAJOR) & BYTE;  /* major device nr */
                task = dmap[major].dmap_task;   /* device task nr */
```

```
                (*dmap[major].dmap_close)(task, &dev_mess);
                rfilp->filp_mode = FILP_CLOSED;
        }
  }
  return(OK);
}


/*===========================================================================*
 *                              do_set                                       *
 *===========================================================================*/
PUBLIC int do_set()
{
/* Set uid_t or gid_t field. */

  register struct fproc *tfp;

  /* Only MM may make this call directly. */
  if (who != MM_PROC_NR) return(EGENERIC);

  tfp = &fproc[slot1];
  if (fs_call == SETUID) {
        tfp->fp_realuid = (uid_t) real_user_id;
        tfp->fp_effuid =  (uid_t) eff_user_id;
  }
  if (fs_call == SETGID) {
        tfp->fp_effgid =  (gid_t) eff_grp_id;
        tfp->fp_realgid = (gid_t) real_grp_id;
  }
  return(OK);
}


/*===========================================================================*
 *                              do_revive                                    *
 *===========================================================================*/
PUBLIC int do_revive()
{
/* A task, typically TTY, has now gotten the characters that were needed for a
 * previous read.  The process did not get a reply when it made the call.
 * Instead it was suspended.  Now we can send the reply to wake it up.  This
 * business has to be done carefully, since the incoming message is from
 * a task (to which no reply can be sent), and the reply must go to a process
 * that blocked earlier.  The reply to the caller is inhibited by setting the
 * 'dont_reply' flag, and the reply to the blocked process is done explicitly
 * in revive().
 */

#if !ALLOW_USER_SEND
  if (who >= LOW_USER) return(EPERM);
#endif

  revive(m.REP_PROC_NR, m.REP_STATUS);
  dont_reply = TRUE;            /* don't reply to the TTY task */
  return(OK);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/device.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* When a needed block is not in the cache, it must be fetched from the disk.
 * Special character files also require I/O.  The routines for these are here.
 *
 * The entry points in this file are:
 *   dev_io:     perform a read or write on a block or character device
 *   dev_opcl:   perform generic device-specific processing for open & close
 *   tty_open:   perform tty-specific processing for open
 *   ctty_open:  perform controlling-tty-specific processing for open
 *   ctty_close: perform controlling-tty-specific processing for close
 *   do_setsid:  perform the SETSID system call (FS side)
 *   do_ioctl:   perform the IOCTL system call
 *   call_task:  procedure that actually calls the kernel tasks
 *   call_ctty:  procedure that actually calls task for /dev/tty
 */

#include "fs.h"
#include <fcntl.h>
#include <minix/callnr.h>
#include <minix/com.h>
#include "dev.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

PRIVATE message dev_mess;
PRIVATE major, minor, task;

FORWARD _PROTOTYPE( void find_dev, (Dev_t dev)                          );

/*===========================================================================*
 *                              dev_io                                       *
 *===========================================================================*/
PUBLIC int dev_io(op, nonblock, dev, pos, bytes, proc, buff)
int op;                         /* DEV_READ, DEV_WRITE, DEV_IOCTL, etc. */
int nonblock;                   /* TRUE if nonblocking op */
dev_t dev;                      /* major-minor device number */
off_t pos;                      /* byte position */
int bytes;                      /* how many bytes to transfer */
int proc;                       /* in whose address space is buff? */
char *buff;                     /* virtual address of the buffer */
{
/* Read or write from a device.  The parameter 'dev' tells which one. */

  find_dev(dev);                /* load the variables major, minor, and task */

  /* Set up the message passed to task. */
  dev_mess.m_type   = op;
  dev_mess.DEVICE   = (dev >> MINOR) & BYTE;
  dev_mess.POSITION = pos;
  dev_mess.PROC_NR  = proc;
  dev_mess.ADDRESS  = buff;
  dev_mess.COUNT    = bytes;
  dev_mess.TTY_FLAGS = nonblock; /* temporary kludge */

```

```
  /* Call the task. */
  (*dmap[major].dmap_rw)(task, &dev_mess);

  /* Task has completed.  See if call completed. */
  if (dev_mess.REP_STATUS == SUSPEND) {
        if (op == DEV_OPEN) task = XPOPEN;
        suspend(task);          /* suspend user */
  }

  return(dev_mess.REP_STATUS);
}


/*===========================================================================*
 *                              dev_opcl                                     *
 *===========================================================================*/
PUBLIC void dev_opcl(task_nr, mess_ptr)
int task_nr;                    /* which task */
message *mess_ptr;              /* message pointer */
{
/* Called from the dmap struct in table.c on opens & closes of special files.*/

  int op;

  op = mess_ptr->m_type;        /* save DEV_OPEN or DEV_CLOSE for later */
  mess_ptr->DEVICE = (mess_ptr->DEVICE >> MINOR) & BYTE;
  mess_ptr->PROC_NR = fp - fproc;

  call_task(task_nr, mess_ptr);

  /* Task has completed.  See if call completed. */
  if (mess_ptr->REP_STATUS == SUSPEND) {
        if (op == DEV_OPEN) task_nr = XPOPEN;
        suspend(task_nr);       /* suspend user */
  }
}

/*===========================================================================*
 *                              tty_open                                     *
 *===========================================================================*/
PUBLIC void tty_open(task_nr, mess_ptr)
int task_nr;
message *mess_ptr;
{
/* This procedure is called from the dmap struct in table.c on tty opens. */

  int r;
  dev_t dev;
  int flags, proc;
  register struct fproc *rfp;

  dev = (dev_t) mess_ptr->DEVICE;
  flags = mess_ptr->COUNT;
  proc = fp - fproc;

  /* Add O_NOCTTY to the flags if this process is not a session leader, or
   * if it already has a controlling tty, or if it is someone elses
   * controlling tty.
   */
  if (!fp->fp_sesldr || fp->fp_tty != 0) {
```

```
                flags |= O_NOCTTY;
        } else {
                for (rfp = &fproc[LOW_USER]; rfp < &fproc[NR_PROCS]; rfp++) {
                        if (rfp->fp_tty == dev) flags |= O_NOCTTY;
                }
        }

        r = dev_io(DEV_OPEN, mode, dev, (off_t) 0, flags, proc, NIL_PTR);

        if (r == 1) {
                fp->fp_tty = dev;
                r = OK;
        }

        mess_ptr->REP_STATUS = r;
}


/*===========================================================================*
 *                              ctty_open                                    *
 *===========================================================================*/
PUBLIC void ctty_open(task_nr, mess_ptr)
int task_nr;
message *mess_ptr;
{
/* This procedure is called from the dmap struct in table.c on opening
 * /dev/tty, the magic device that translates to the controlling tty.
 */

  mess_ptr->REP_STATUS = fp->fp_tty == 0 ? ENXIO : OK;
}


/*===========================================================================*
 *                              ctty_close                                   *
 *===========================================================================*/
PUBLIC void ctty_close(task_nr, mess_ptr)
int task_nr;
message *mess_ptr;
{
/* Close /dev/tty. */

  mess_ptr->REP_STATUS = OK;
}


/*===========================================================================*
 *                              do_setsid                                    *
 *===========================================================================*/
PUBLIC int do_setsid()
{
/* Perform the FS side of the SETSID call, i.e. get rid of the controlling
 * terminal of a process, and make the process a session leader.
 */
  register struct fproc *rfp;

  /* Only MM may do the SETSID call directly. */
  if (who != MM_PROC_NR) return(ENOSYS);

  /* Make the process a session leader with no controlling tty. */
```

```
  rfp = &fproc[slot1];
  rfp->fp_sesldr = TRUE;
  rfp->fp_tty = 0;
}


/*===========================================================================*
 *                              do_ioctl                                     *
 *===========================================================================*/
PUBLIC int do_ioctl()
{
/* Perform the ioctl(ls_fd, request, argx) system call (uses m2 fmt). */

  struct filp *f;
  register struct inode *rip;
  dev_t dev;

  if ( (f = get_filp(ls_fd)) == NIL_FILP) return(err_code);
  rip = f->filp_ino;            /* get inode pointer */
  if ( (rip->i_mode & I_TYPE) != I_CHAR_SPECIAL
        && (rip->i_mode & I_TYPE) != I_BLOCK_SPECIAL) return(ENOTTY);
  dev = (dev_t) rip->i_zone[0];
  find_dev(dev);

  dev_mess= m;

  dev_mess.m_type  = DEV_IOCTL;
  dev_mess.PROC_NR = who;
  dev_mess.TTY_LINE = minor;

  /* Call the task. */
  (*dmap[major].dmap_rw)(task, &dev_mess);

  /* Task has completed.  See if call completed. */
  if (dev_mess.REP_STATUS == SUSPEND) {
        if (f->filp_flags & O_NONBLOCK) {
                /* Not supposed to block. */
                dev_mess.m_type = CANCEL;
                dev_mess.PROC_NR = who;
                dev_mess.TTY_LINE = minor;
                (*dmap[major].dmap_rw)(task, &dev_mess);
                if (dev_mess.REP_STATUS == EINTR) dev_mess.REP_STATUS = EAGAIN;
        } else {
                suspend(task);          /* User must be suspended. */
        }
  }
  return(dev_mess.REP_STATUS);
}


/*===========================================================================*
 *                              find_dev                                     *
 *===========================================================================*/
PRIVATE void find_dev(dev)
dev_t dev;                      /* device */
{
/* Extract the major and minor device number from the parameter. */

  major = (dev >> MAJOR) & BYTE;        /* major device number */
  minor = (dev >> MINOR) & BYTE;        /* minor device number */
```

```
  if (major >= max_major) {
        major = minor = 0;              /* will fail with ENODEV */
  }
  task = dmap[major].dmap_task; /* which task services the device */
}


/*===========================================================================*
 *                              call_task                                    *
 *===========================================================================*/
PUBLIC void call_task(task_nr, mess_ptr)
int task_nr;                    /* which task to call */
message *mess_ptr;              /* pointer to message for task */
{
/* All file system I/O ultimately comes down to I/O on major/minor device
 * pairs.  These lead to calls on the following routines via the dmap table.
 */

  int r, proc_nr;
  message local_m;

  proc_nr = mess_ptr->PROC_NR;

  while ((r = sendrec(task_nr, mess_ptr)) == ELOCKED) {
        /* sendrec() failed to avoid deadlock. The task 'task_nr' is
         * trying to send a REVIVE message for an earlier request.
         * Handle it and go try again.
         */
        if ((r = receive(task_nr, &local_m)) != OK) break;

        /* If we're trying to send a cancel message to a task which has just
         * sent a completion reply, ignore the reply and abort the cancel
         * request. The caller will do the revive for the process.
         */
        if (mess_ptr->m_type == CANCEL && local_m.REP_PROC_NR == proc_nr)
                return;

        /* Otherwise it should be a REVIVE. */
        if (local_m.m_type != REVIVE) {
                printf(
                "fs: strange device reply from %d, type = %d, proc = %d\n",
                        local_m.m_source,
                        local_m.m_type, local_m.REP_PROC_NR);
                continue;
        }

        revive(local_m.REP_PROC_NR, local_m.REP_STATUS);
  }

  /* The message received may be a reply to this call, or a REVIVE for some
   * other process.
   */
  for (;;) {
        if (r != OK) panic("call_task: can't send/receive", NO_NUM);

        /* Did the process we did the sendrec() for get a result? */
        if (mess_ptr->REP_PROC_NR == proc_nr) break;

        /* Otherwise it should be a REVIVE. */
        if (mess_ptr->m_type != REVIVE) {
```

```
                printf(
                "fs: strange device reply from %d, type = %d, proc = %d\n",
                        mess_ptr->m_source,
                        mess_ptr->m_type, mess_ptr->REP_PROC_NR);
                continue;
        }
        revive(mess_ptr->REP_PROC_NR, mess_ptr->REP_STATUS);

        r = receive(task_nr, mess_ptr);
  }
}


/*===========================================================================*
 *                              call_ctty                                    *
 *===========================================================================*/
PUBLIC void call_ctty(task_nr, mess_ptr)
int task_nr;                    /* not used - for compatibility with dmap_t */
message *mess_ptr;              /* pointer to message for task */
{
/* This routine is only called for one device, namely /dev/tty.  Its job
 * is to change the message to use the controlling terminal, instead of the
 * major/minor pair for /dev/tty itself.
 */

  int major_device;

  if (fp->fp_tty == 0) {
        /* No controlling tty present anymore, return an I/O error. */
        mess_ptr->REP_STATUS = EIO;
        return;
  }
  major_device = (fp->fp_tty >> MAJOR) & BYTE;
  task_nr = dmap[major_device].dmap_task;       /* task for controlling tty */
  mess_ptr->DEVICE = (fp->fp_tty >> MINOR) & BYTE;
  call_task(task_nr, mess_ptr);
}


/*===========================================================================*
 *                              no_dev                                       *
 *===========================================================================*/
PUBLIC void no_dev(task_nr, m_ptr)
int task_nr;                    /* not used - for compatibility with dmap_t */
message *m_ptr;                 /* message pointer */
{
/* No device there. */

  m_ptr->REP_STATUS = ENODEV;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                 src/fs/utility.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

/* This file contains a few general purpose utility routines.
 *
 * The entry points into this file are
 *   clock_time:  ask the clock task for the real time
 *   copy:        copy a block of data
 *   fetch_name:  go get a path name from user space
 *   no_sys:      reject a system call that FS does not handle
 *   panic:       something awful has occurred;  MINIX cannot continue
 *   conv2:       do byte swapping on a 16-bit int
 *   conv4:       do byte swapping on a 32-bit long
 */

#include "fs.h"
#include <minix/com.h>
#include <minix/boot.h>
#include <unistd.h>
#include "buf.h"
#include "file.h"
#include "fproc.h"
#include "inode.h"
#include "param.h"

PRIVATE int panicking;          /* inhibits recursive panics during sync */
PRIVATE message clock_mess;

/*===========================================================================*
 *                              clock_time                                   *
 *===========================================================================*/
PUBLIC time_t clock_time()
{
/* This routine returns the time in seconds since 1.1.1970.  MINIX is an
 * astrophysically naive system that assumes the earth rotates at a constant
 * rate and that such things as leap seconds do not exist.
 */

  register int k;

  clock_mess.m_type = GET_TIME;
  if ( (k = sendrec(CLOCK, &clock_mess)) != OK) panic("clock_time err", k);

  return( (time_t) clock_mess.NEW_TIME);
}


/*===========================================================================*
 *                              fetch_name                                   *
 *===========================================================================*/
PUBLIC int fetch_name(path, len, flag)
char *path;                     /* pointer to the path in user space */
int len;                        /* path length, including 0 byte */
int flag;                       /* M3 means path may be in message */
{
/* Go get path and put it in 'user_path'.
 * If 'flag' = M3 and 'len' <= M3_STRING, the path is present in 'message'.
 * If it is not, go copy it from user space.
```

```
 */

  register char *rpu, *rpm;
  int r;

  /* Check name length for validity. */
  if (len <= 0) {
        err_code = EINVAL;
        return(EGENERIC);
  }

  if (len > PATH_MAX) {
        err_code = ENAMETOOLONG;
        return(EGENERIC);
  }

  if (flag == M3 && len <= M3_STRING) {
        /* Just copy the path from the message to 'user_path'. */
        rpu = &user_path[0];
        rpm = pathname;         /* contained in input message */
        do { *rpu++ = *rpm++; } while (--len);
        r = OK;
  } else {
        /* String is not contained in the message.  Get it from user space. */
        r = sys_copy(who, D, (phys_bytes) path,
                FS_PROC_NR, D, (phys_bytes) user_path, (phys_bytes) len);
  }
  return(r);
}


/*===========================================================================*
 *                              no_sys                                       *
 *===========================================================================*/
PUBLIC int no_sys()
{
/* Somebody has used an illegal system call number */

  return(EINVAL);
}


/*===========================================================================*
 *                              panic                                        *
 *===========================================================================*/
PUBLIC void panic(format, num)
char *format;                   /* format string */
int num;                        /* number to go with format string */
{
/* Something awful has happened.  Panics are caused when an internal
 * inconsistency is detected, e.g., a programming error or illegal value of a
 * defined constant.
 */

  if (panicking) return;        /* do not panic during a sync */
  panicking = TRUE;             /* prevent another panic during the sync */
  printf("File system panic: %s ", format);
  if (num != NO_NUM) printf("%d",num);
  printf("\n");
  (void) do_sync();             /* flush everything to the disk */
```

```
  sys_abort(RBT_PANIC);
}


/*===========================================================================*
 *                              conv2                                        *
 *===========================================================================*/
PUBLIC unsigned conv2(norm, w)
int norm;                       /* TRUE if no swap, FALSE for byte swap */
int w;                          /* promotion of 16-bit word to be swapped */
{
/* Possibly swap a 16-bit word between 8086 and 68000 byte order. */

  if (norm) return( (unsigned) w & 0xFFFF);
  return( ((w&BYTE) << 8) | ( (w>>8) & BYTE));
}


/*===========================================================================*
 *                              conv4                                        *
 *===========================================================================*/
PUBLIC long conv4(norm, x)
int norm;                       /* TRUE if no swap, FALSE for byte swap */
long x;                         /* 32-bit long to be byte swapped */
{
/* Possibly swap a 32-bit long between 8086 and 68000 byte order. */

  unsigned lo, hi;
  long l;

  if (norm) return(x);                  /* byte order was already ok */
  lo = conv2(FALSE, (int) x & 0xFFFF);  /* low-order half, byte swapped */
  hi = conv2(FALSE, (int) (x>>16) & 0xFFFF);    /* high-order half, swapped */
  l = ( (long) lo <<16) | hi;
  return(l);
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                src/fs/putk.c
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
/* FS must occasionally print some message.  It uses the standard library
 * routine prink().  (The name "printf" is really a macro defined as "printk").
 * Printing is done by calling the TTY task directly, not going through FS.
 */

#include "fs.h"
#include <minix/com.h>

#define BUF_SIZE          100   /* print buffer size */

PRIVATE int buf_count;          /* # characters in the buffer */
PRIVATE char print_buf[BUF_SIZE];       /* output is buffered here */
PRIVATE message putch_msg;      /* used for message to TTY task */

FORWARD _PROTOTYPE( void flush, (void)                                  );
```

```
/*===========================================================================*
 *                              putk                                         *
 *===========================================================================*/
PUBLIC void putk(c)
int c;
{
/* Accumulate another character.  If 0 or buffer full, print it. */

  if (c == 0 || buf_count == BUF_SIZE) flush();
  if (c == '\n') putk('\r');
  if (c != 0) print_buf[buf_count++] = c;
}


/*===========================================================================*
 *                              flush                                        *
 *===========================================================================*/
PRIVATE void flush()
{
/* Flush the print buffer by calling TTY task. */


  if (buf_count == 0) return;
  putch_msg.m_type = DEV_WRITE;
  putch_msg.PROC_NR  = 1;
  putch_msg.TTY_LINE = 0;
  putch_msg.ADDRESS  = print_buf;
  putch_msg.COUNT = buf_count;
  call_task(TTY, &putch_msg);
  buf_count = 0;
}
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                                  ./end_of_list
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```