NAME

execve − execute a file

SYNOPSIS

```
#include <unistd.h>

int execve(const char *name, char *const argv[], char *const envp[])
```

DESCRIPTION

Execve transforms the calling process into a new process. The new process is constructed from an ordinary file called the *new process file*. This file is either an executable object file, or a file of data for an interpreter. An executable object file consists of an identifying header, followed by pages of data representing the initial program (text) and initialized data pages. Additional pages may be specified by the header to be initialized with zero data. See **a.out**(5).

An interpreter file begins with a line of the form ‘‘#! *interpreter*’’. When an interpreter file is **execve** ’d, the system **execve** ’s the specified *interpreter*, giving it the name of the originally exec’d file as an argument and shifting over the rest of the original arguments.

There can be no return from a successful **execve** because the calling core image is lost. This is the mechanism whereby different process images become active.

The argument *argv* is a null-terminated array of character pointers to null-terminated character strings. These strings constitute the argument list to be made available to the new process. By convention, at least one argument must be present in this array, and the first element of this array should be the name of the executed program (i.e., the last component of *name*).

The argument *envp* is also a null-terminated array of character pointers to null-terminated strings. These strings pass information to the new process that is not directly an argument to the command (see **environ**(7)).

Descriptors open in the calling process remain open in the new process, except for those for which the close-on-exec flag is set (see **close**(2)). Descriptors that remain open are unaffected by **execve**.

Ignored signals remain ignored across an **execve**, but signals that are caught are reset to their default values. Blocked signals remain blocked regardless of changes to the signal action. The signal stack is reset to be undefined (see **sigaction**(2) for more information).

Each process has *real* user and group IDs and an *effective* user and group IDs. The *real* ID identifies the person using the system; the *effective* ID determines his access privileges. **Execve** changes the effective user and group ID to the owner of the executed file if the file has the set-user-ID or set-group-ID modes. The *real* user ID is not affected.

The new process also inherits the following attributes from the calling process:

process ID	see **getpid** (2)
parent process ID	see **getppid** (2)
process group ID	see **getpgrp** (2)
access groups	see **getgroups** (2)
working directory	see **chdir** (2)
root directory	see **chroot** (2)
control terminal	see **tty** (4)
alarm timer	see **alarm** (2)
file mode mask	see **umask** (2)
signal mask	see **sigaction** (2), **sigprocmask** (2)

When the executed program begins, it is called as follows:

```
int main(int argc, char *const argv[], char *const envp[]);

exit(main(argc, argv, envp));
```

where *argc* is the number of elements in *argv* (the ‘‘arg count’’) and *argv* is the array of character pointers to the arguments themselves.

Envp is a pointer to an array of strings that constitute the *environment* of the process. A pointer to this array is also stored in the global variable ‘‘environ’’. Each string consists of a name, an =, and a null-terminated value. The array of pointers is terminated by a null pointer. The shell **sh**(1) passes an environment entry for each global shell variable defined when the program is called. See **environ**(7) for some conventionally used names.

RETURN VALUE

If **execve** returns to the calling process an error has occurred; the return value will be −1 and the global variable **errno** will contain an error code.

ERRORS

Execve will fail and return to the calling process if one or more of the following are true:

[ENOTDIR] A component of the path prefix is not a directory.

[ENAMETOOLONG] The path name exceeds PATH_MAX characters.

[ENOENT] The new process file does not exist.

[ELOOP] Too many symbolic links were encountered in translating the pathname. (Minix-vmd)

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] The new process file is not an ordinary file.

[EACCES] The new process file mode denies execute permission.

[ENOEXEC] The new process file has the appropriate access permission, but has an invalid magic number in its header.

[ENOMEM] The new process requires more (virtual) memory than is currently available.

[E2BIG] The number of bytes in the new process’s argument list is larger than the system-imposed limit ARG_MAX. The limit in the system as released is 4096 bytes for 16-bit Minix, 16384 bytes for 32-bit Minix, and unlimited for Minix-vmd.

[EFAULT] *Path* , *argv* , or *envp* point to an illegal address.

[EIO] An I/O error occurred while reading from the file system.

CAVEATS

If a program is *setuid* to a non-super-user, but is executed when the real **uid** is ‘‘root’’, then the program has some of the powers of a super-user as well.

SEE ALSO

exit(2), **fork**(2), **execl**(3), **environ**(7).