NAME

exec, execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

#include <unistd.h>

int execl(const char ****path*, const char ****arg0***, . . ., const char ****argn***, char *** /*NULL*/**);**

int execv(const char ****path*, char *const** *argv[ ]***);**

int execle(const char ****path*,char *const** *arg0[ ]***, . . . , const char ****argn***,**
char * /*NULL*/**, char *const** *envp[ ]***);**

int execve (const char ****path*, char *const** *argv[ ]* **char *const** *envp[ ]***);**

int execlp (const char * *file***, const char ****arg0***, . . ., const char ****argn***, char *** /*NULL*/**);**

int execvp (const char * *file***, char *const** *argv[ ]***);**

DESCRIPTION

Each of the functions in the **exec** family overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

When a C program is executed, it is called as follows:

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

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is at least one, and the first member of the array points to a string containing the name of the file.

The arguments *arg0***,** . . .**,** *argn* point to null-terminated character strings. These strings constitute the argument list available to the new process image. Conventionally at least *arg0* should be present. The *arg0* argument points to a string that is the same as *path* (or the last component of *path*). The list of argument strings is terminated by a **(char** ∗**)0** argument.

The *argv* argument is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process image. By convention, *argv* must have at least one member, and it should point to a string that is the same as *path* (or its last component). The *argv* argument is terminated by a null pointer.

The *path* argument points to a path name that identifies the new process file.

The *file* argument points to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the **PATH** environment variable (see **environ**(5)).

File descriptors open in the calling process remain open in the new process.

Signals that are being caught by the calling process are set to the default disposition in the new process image (see **signal**(3C)). Otherwise, the new process image inherits the signal dispositions of the calling process.

RETURN VALUES

If a function in the **exec** family returns to the calling process, an error has occurred; the return value is **–1** and **errno** is set to indicate the error.

NAME

fork – create a child process

SYNOPSIS

#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID.
* The child's parent process ID is the same as the parent's process ID.

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, −1 is returned in the parent, no child process is created, and *errno* is set appropriately.

ERRORS

EAGAIN

A system-imposed limit on the number of threads was encountered. There are a number of limits that may trigger this error:

* the **RLIMIT_NPROC** soft resource limit (set via **setrlimit**(2)), which limits the number of processes and threads for a real user ID, was reached;
* the kernel's system-wide limit on the number of processes and threads, */proc/sys/kernel/threads-max*, was reached (see **proc**(5));
* the maximum number of PIDs, */proc/sys/kernel/pid_max*, was reached (see **proc**(5)); or
* the PID limit (*pids.max*) imposed by the cgroup "process number" (PIDs) controller was reached.

EAGAIN

The caller is operating under the **SCHED_DEADLINE** scheduling policy and does not have the reset-on-fork flag set. See **sched**(7).

ENOMEM

fork() failed to allocate the necessary kernel structures because memory is tight.

ENOMEM

An attempt was made to create a child process in a PID namespace whose "init" process has terminated. See **pid_namespaces**(7).

ENOSYS

fork() is not supported on this platform (for example, hardware without a Memory-Management Unit).

ERESTARTNOINTR (since Linux 2.6.17)

System call was interrupted by a signal and will be restarted. (This can be seen only during a trace.)

NAME

perror – print a system error message

SYNOPSIS

#include <stdio.h>

void perror(const char ****s*);**

#include <errno.h>

const char * const *sys_errlist*[];
int *sys_nerr*;
int *errno*;

DESCRIPTION

The **perror**() function produces a message on standard error describing the last error encountered during a call to a system or library function.

First (if *s* is not NULL and **s* is not a null byte ('\0')), the argument string *s* is printed, followed by a colon and a blank. Then an error message corresponding to the current value of *errno* and a new-line.

To be of most use, the argument string should include the name of the function that incurred the error.

The global error list *sys_errlist*[], which can be indexed by *errno*, can be used to obtain the error message without the newline. The largest message number provided in the table is *sys_nerr*−1. Be careful when directly accessing this list, because new error values may not have been added to *sys_errlist*[]. The use of *sys_errlist*[] is nowadays deprecated; use **strerror**(3) instead.

When a system call fails, it usually returns −1 and sets the variable *errno* to a value describing what went wrong. (These values can be found in <*errno.h*>.) Many library functions do likewise. The function **perror**() serves to translate this error code into human-readable form. Note that *errno* is undefined after a successful system call or library function call: this call may well change this variable, even though it succeeds, for example because it internally used some other library function that failed. Thus, if a failing call is not immediately followed by a call to **perror**(), the value of *errno* should be saved.

SEE ALSO

err(3), **errno**(3), **error**(3), **strerror**(3)

NAME

stat, fstat, lstat – get file status

SYNOPSIS

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char ****path*, struct stat ****buf***);**
int fstat(int *fd***, struct stat ****buf***);**
int lstat(const char ****path*, struct stat ****buf***);**

DESCRIPTION

These functions return information about a file. No permissions are required on the file itself, but — in the case of **stat**() and **lstat**() — execute (search) permission is required on all of the directories in *path* that lead to the file.

stat() stats the file pointed to by *path* and fills in *buf*.

lstat() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to **stat**(), except that the file to be stat-ed is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file  */
    ino_t     st_ino;     /* inode number                  */
    mode_t    st_mode;    /* protection                    */
    nlink_t   st_nlink;   /* number of hard links          */
    ...
    off_t     st_size;    /* total size, in bytes          */
    blksize_t st_blksize; /* blocksize for file system I/O */
    ...
};
```

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EACCES
Search permission is denied for one of the directories in the path prefix of *path*. (See also **path_resolution**(7).)

...

ENAMETOOLONG
File name too long.

ENOENT
A component of the path *path* does not exist, or the path is an empty string.

ENOMEM
Out of memory (i.e., kernel memory).

NAME

wait, waitpid – wait for process to change state

SYNOPSIS

#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int ****wstatus*);**

pid_t waitpid(pid_t *pid***, int ****wstatus***, int** *options***);**

DESCRIPTION

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of **sigaction**(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

wait() and waitpid()

The **wait**() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

```
waitpid(−1, &wstatus, 0);
```

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

< −1 meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.

−1 meaning wait for any child process.

0 meaning wait for any child process whose process group ID is equal to that of the calling process.

> 0 meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG return immediately if no child has exited.

WUNTRACED
also return if a child has stopped (but not traced via **ptrace**(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)
also return if a stopped child has been resumed by delivery of **SIGCONT**.

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and **waitpid**()!):

WIFEXITED(*wstatus***)**
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(*wstatus***)**
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to **exit**(3) or **_exit**(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

WIFSIGNALED(*wstatus***)**
returns true if the child process was terminated by a signal.

WTERMSIG(*wstatus***)**
returns the number of the signal that caused the child process to terminate. This macro should be employed only if **WIFSIGNALED** returned true.

WCOREDUMP(*wstatus***)**
returns true if the child produced a core dump. This macro should be employed only if **WIFSIGNALED** returned true.

This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementations (e.g., AIX, SunOS). Therefore, enclose its use inside *#ifdef WCOREDUMP ... #endif*.

WIFSTOPPED(*wstatus***)**
returns true if the child process was stopped by delivery of a signal; this is possible only if the call was done using **WUNTRACED** or when the child is being traced (see **ptrace**(2)).

WSTOPSIG(*wstatus***)**
returns the number of the signal which caused the child to stop. This macro should be employed only if **WIFSTOPPED** returned true.

WIFCONTINUED(*wstatus***)**
(since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

RETURN VALUE

wait(): on success, returns the process ID of the terminated child; on error, −1 is returned.

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On error, −1 is returned.

Each of these calls sets *errno* to an appropriate value in the case of an error.

ERRORS

ECHILD
(for **wait**()) The calling process does not have any unwaited-for children.

ECHILD
(for **waitpid**() or **waitid**()) The process specified by *pid* (**waitpid**()) or *idtype* and *id* (**waitid**()) does not exist or is not a child of the calling process. (This can happen for one's own child if the action for **SIGCHLD** is set to **SIG_IGN**. See also the *Linux Notes* section about threads.)

EINTR
WNOHANG was not set and an unblocked signal or a **SIGCHLD** was caught; see **signal**(7).

EINVAL
The *options* argument was invalid.