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

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

DESCRIPTION

These functions return information about the specified file. You do not need any access rights to the file to get this information but you need search rights to all directories named in the path leading to the file.

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

lstat is identical to **stat**, except in the case of a symbolic link, where the link itself is stat-ed, not the file that it refers to.

They all return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t       st_dev;      /* device */
    ino_t       st_ino;      /* inode */
    mode_t      st_mode;     /* protection */
    nlink_t     st_nlink;    /* number of hard links */
    uid_t       st_uid;      /* user ID of owner */
    gid_t       st_gid;      /* group ID of owner */
    dev_t       st_rdev;     /* device type (if inode device) */
    off_t       st_size;     /* total size, in bytes */
    blksize_t   st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t    st_blocks;   /* number of blocks allocated */
    time_t      st_atime;    /* time of last access */
    time_t      st_mtime;    /* time of last modification */
    time_t      st_ctime;    /* time of last status change */
};
```

The value *st_size* gives the size of the file (if it is a regular file or a symlink) in bytes. The size of a symlink is the length of the pathname it contains, without trailing NUL.

The following POSIX macros are defined to check the file type in the field *st_mode:*

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?

RETURN VALUE

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

ERRORS

EACCES	Search permisson is denied for one of the directories in the path prefix of *path*.
ENOENT	A component of *path* does not exist, or *path* is an empty string.
ENOTDIR	A component of the path prefix of *path* is not a directory.

NAME

waitpid − wait for child process to change state

SYNOPSIS

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

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

DESCRIPTION

waitpid( ) suspends the calling process until one of its children changes state; if a child process changed state prior to the call to **waitpid( )**, return is immediate. *pid* specifies a set of child processes for which status is requested.

If *pid* is equal to **(pid_t)−1**, status is requested for any child process.

If *pid* is greater than **(pid_t)0**, it specifies the process ID of the child process for which status is requested.

If *pid* is equal to **(pid_t)0** status is requested for any child process whose process group ID is equal to that of the calling process.

If *pid* is less than **(pid_t)−1**, status is requested for any child process whose process group ID is equal to the absolute value of *pid*.

If **waitpid( )** returns because the status of a child process is available, then that status may be evaluated with the macros defined by **wstat**(5)**.** If the calling process had specified a non-zero value of *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*.

The *options* argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header **<sys/wait.h>**:

WCONTINUED The status of any continued child process specified by *pid*, whose status has not been reported since it continued, is also reported to the calling process.

WNOHANG **waitpid( )** will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.

WNOWAIT Keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with identical results.

RETURN VALUES

If **waitpid( )** returns because the status of a child process is available, this function returns a value equal to the process ID of the child process for which status is reported. If **waitpid( )** returns due to the delivery of a signal to the calling process, **−1** is returned and **errno** is set to **EINTR**. If this function was invoked with **WNOHANG** set in *options*, it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, **0** is returned. Otherwise, **−1** is returned, and **errno** is set to indicate the error.

ERRORS

waitpid( ) will fail if one or more of the following is true:

ECHILD The process or process group specified by *pid* does not exist or is not a child of the calling process or can never be in the states specified by *options*.

EINTR **waitpid( )** was interrupted due to the receipt of a signal sent by the calling process.

EINVAL An invalid value was specified for *options*.

SEE ALSO

exec(2), **exit**(2), **fork**(2), **sigaction**(2), **wstat**(5)