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 <unistd.h>

pid_t fork(void);

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process, referred to as the *child*, is an exact duplicate of the calling process, referred to as the *parent*, except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group (**setpgid**(2)).

* The child's parent process ID is the same as the parent's process ID.

* The child does not inherit its parent's memory locks (**mlock**(2), **mlockall**(2)).

* Process resource utilizations (**getrusage**(2)) and CPU time counters (**times**(2)) are reset to zero in the child.

* The child's set of pending signals is initially empty (**sigpending**(2)).

* The child does not inherit semaphore adjustments from its parent (**semop**(2)).

* The child does not inherit record locks from its parent (**fcntl**(2)).

* The child does not inherit timers from its parent (**setitimer**(2), **alarm**(2), **timer_create**(2)).

* The child does not inherit outstanding asynchronous I/O operations from its parent (**aio_read**(3), **aio_write**(3)), nor does it inherit any asynchronous I/O contexts from its parent (see **io_setup**(2)).

Note the following further points:

* The child process is created with a single thread — the one that called **fork**(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects; the use of **pthread_atfork**(3) may be helpful for dealing with problems that this can cause.

* The child inherits copies of the parent's set of open file descriptors. Each file descriptor in the child refers to the same open file description (see **open**(2)) as the corresponding file descriptor in the parent. This means that the two descriptors share open file status flags, current file offset, and signal-driven I/O attributes (see the description of **F_SETOWN** and **F_SETSIG** in **fcntl**(2)).

* The child inherits copies of the parent's set of open directory streams (see **opendir**(3)). POSIX.1-2001 says that the corresponding directory streams in the parent and child *may* share the directory stream positioning; on Linux/glibc they do not.

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

fork() cannot allocate sufficient memory to copy the parent's page tables and allocate a task structure for the child.

EAGAIN

It was not possible to create a new process because the caller's **RLIMIT_NPROC** resource limit was encountered. To exceed this limit, the process must have either the **CAP_SYS_ADMIN** or the **CAP_SYS_RESOURCE** capability.

ENOMEM

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

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf − formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(const char * format, ...);
int fprintf(FILE *stream, const char * format, ...);
int sprintf(char *str, const char * format, ...);
int snprintf(char *str, size_t size, const char * format, ...);

 ...
```

DESCRIPTION

The functions in the **printf**() family produce output according to a *format* as described below. The functions **printf**() and **vprintf**() write output to *stdout*, the standard output stream; **fprintf**() and **vfprintf**() write output to the given output *stream*; **sprintf**(), **snprintf**(), **vsprintf**() and **vsnprintf**() write to the character string *str*.

The functions **snprintf**() and **vsnprintf**() write at most *size* bytes (including the trailing null byte ('\0')) to *str*.

The functions **vprintf**(), **vfprintf**(), **vsprintf**(), **vsnprintf**() are equivalent to the functions **printf**(), **fprintf**(), **sprintf**(), **snprintf**(), respectively, except that they are called with a *va_list* instead of a variable number of arguments. These functions do not call the *va_end* macro. Because they invoke the *va_arg* macro, the value of *ap* is undefined after the call. See **stdarg**(3).

These eight functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of **stdarg**(3)) are converted for output.

Return value

Upon successful return, these functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

The functions **snprintf**() and **vsnprintf**() do not write more than *size* bytes (including the trailing '\0'). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. Thus, a return value of *size* or more means that the output was truncated. (See also below under NOTES.)

If an output error is encountered, a negative value is returned.

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*. In between there may be (in this order) zero or more *flags*, an optional minimum *field width*, an optional *precision* and an optional *length modifier*.

The arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each '*' and each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given). One can also specify explicitly which argument is taken, at each place where an argument is required, by writing "%m$" instead of '%' and "*m$" instead of '*', where the decimal integer m denotes the position in the argument list of the desired argument, indexed starting from 1. Thus,

```
printf("%*d", width, num);
```

and

```
printf("%2$*1$d", width, num);
```

are equivalent. The second style allows repeated references to the same argument. The C99 standard does not include the style using '$', which comes from the Single Unix Specification. If the style using '$' is used, it must be used throughout for all conversions taking an argument and all width and precision arguments, but it may be mixed with "%%" formats which do not consume an argument. There may be no gaps in the numbers of arguments specified using '$'; for example, if arguments 1 and 3 are specified, argument 2 must also be specified somewhere in the format string.

For some numeric conversions a radix character ("decimal point") or thousands' grouping character is used. The actual character used depends on the **LC_NUMERIC** part of the locale. The POSIX locale uses '.' as radix character, and does not have a grouping character. Thus,

```
printf("%'.2f", 1234567.89);
```

results in "1234567.89" in the POSIX locale, in "1234567,89" in the nl_NL locale, and in "1.234.567,89" in the da_DK locale.

The conversion specifier

A character that specifies the type of conversion to be applied. An example for a conversion specifier is:

s The *const char ** argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

SEE ALSO

printf(1), **asprintf**(3), **dprintf**(3), **scanf**(3), **setlocale**(3), **wcrtomb**(3), **wprintf**(3), **locale**(5)

COLOPHON

This page is part of release 3.05 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

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)