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)).

The process attributes in the preceding list are all specified in POSIX.1-2001. The parent and child also differ with respect to the following Linux-specific process attributes:

* The child does not inherit directory change notifications (dnotify) from its parent (see the description of **F_NOTIFY** in **fcntl**(2)).
* The **prctl**(2) **PR_SET_PDEATHSIG** setting is reset so that the child does not receive a signal when its parent terminates.
* Memory mappings that have been marked with the **madvise**(2) **MADV_DONTFORK** flag are not inherited across a **fork**().
* The termination signal of the child is always **SIGCHLD** (see **clone**(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 message queue descriptors (see **mq_overview**(7)). Each descriptor in the child refers to the same open message queue description as the corresponding descriptor in the parent. This means that the two descriptors share the same flags (*mq_flags*).
* 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.

CONFORMING TO

SVr4, 4.3BSD, POSIX.1-2001.

NOTES

Under Linux, **fork**() is implemented using copy-on-write pages, so the only penalty that it incurs is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

Since version 2.3.3, rather than invoking the kernel's **fork**() system call, the glibc **fork**() wrapper that is provided as part of the NPTL threading implementation invokes **clone**(2) with flags that provide the same effect as the traditional system call. The glibc wrapper invokes any fork handlers that have been established using **pthread_atfork**(3).

EXAMPLE

See **pipe**(2) and **wait**(2).

SEE ALSO

clone(2), **execve**(2), **setrlimit**(2), **unshare**(2), **vfork**(2), **wait**(2), **daemon**(3), **capabilities**(7), **credentials**(7)

COLOPHON

This page is part of release 3.27 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

opendir − open a directory / readdir − read a directory

SYNOPSIS

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

DIR *opendir(const char * *name*);
int closedir(DIR * *dirp*);
struct dirent *readdir(DIR * *dir*);

DESCRIPTION opendir

The **opendir()** function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE

The **opendir**() function returns a pointer to the directory stream. On error, NULL is returned, and *errno* is set appropriately.

DESCRIPTION closedir

The **closedir()** function closes the directory stream associated with *dirp*. A successful call to **closedir()** also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirpP is not available after this call.*

RETURN VALUE

The **closedir()** function returns 0 on success. On error, −1 is returned, and *errno* is set appropriately.

DESCRIPTION readdir

The **readdir()** function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by *dir*. It returns NULL on reaching the end-of-file or if an error occurred. It is safe to use **readdir()** inside threads if the pointers passed as *dir* are created by distinct calls to **opendir()**.

The data returned by **readdir()** is overwritten by subsequent calls to **readdir()** for the **same** directory stream.

The *dirent* structure is defined as follows:

```
struct dirent {
    long        d_ino;      /* inode number */
    char        d_name[256]; /* filename */
};
```

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure. (This structure may be statically allocated; do not attempt to **free**(3) it.)

If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set appropriately. To distinguish end of stream and from an error, set *errno* to zero before calling **readdir**() and then check the value of *errno* if NULL is returned.

ERRORS

EACCES

Permission denied.

ENOENT

Directory does not exist, or *name* is an empty string.

ENOTDIR

name is not a directory.

NAME

pthread_create – create a new thread / pthread_exit – terminate the calling thread

SYNOPSIS

#include <pthread.h>

int pthread_create(pthread_t * *thread*, **pthread_attr_t** * *attr*, **void** * (**start_routine*)(**void** *), **void** * *arg*);

void pthread_exit(void **retval*);

DESCRIPTION

pthread_create creates a new thread of control that executes concurrently with the calling thread. The new thread applies the function *start_routine* passing it *arg* as first argument. The new thread terminates either explicitly, by calling **pthread_exit**(3), or implicitly, by returning from the *start_routine* function. The latter case is equivalent to calling **pthread_exit**(3) with the result returned by *start_routine* as exit code.

The *attr* argument specifies thread attributes to be applied to the new thread. See **pthread_attr_init**(3) for a complete list of thread attributes. The *attr* argument can also be **NULL**, in which case default attributes are used: the created thread is joinable (not detached) and has default (non real-time) scheduling policy.

pthread_exit terminates the execution of the calling thread. All cleanup handlers that have been set for the calling thread with **pthread_cleanup_push**(3) are executed in reverse order (the most recently pushed handler is executed first). Finalization functions for thread-specific data are then called for all keys that have non- **NULL** values associated with them in the calling thread (see **pthread_key_create**(3)). Finally, execution of the calling thread is stopped.

The *retval* argument is the return value of the thread. It can be consulted from another thread using **pthread_join**(3).

RETURN VALUE

On success, the identifier of the newly created thread is stored in the location pointed by the *thread* argument, and a 0 is returned. On error, a non-zero error code is returned.

The **pthread_exit** function never returns.

ERRORS

EAGAIN
not enough system resources to create a process for the new thread.

EAGAIN
more than **PTHREAD_THREADS_MAX** threads are already active.

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_join(3), **pthread_detach**(3), **pthread_attr_init**(3).

NAME

pthread_detach – put a running thread in the detached state

SYNOPSIS

#include <pthread.h>

int pthread_detach(pthread_t th);

DESCRIPTION

pthread_detach put the thread *th* in the detached state. This guarantees that the memory resources consumed by *th* will be freed immediately when *th* terminates. However, this prevents other threads from synchronizing on the termination of *th* using **pthread_join**.

A thread can be created initially in the detached state, using the **detachstate** attribute to **pthread_create**(3). In contrast, **pthread_detach** applies to threads created in the joinable state, and which need to be put in the detached state later.

After **pthread_detach** completes, subsequent attempts to perform **pthread_join** on *th* will fail. If another thread is already joining the thread *th* at the time **pthread_detach** is called, **pthread_detach** does nothing and leaves *th* in the joinable state.

RETURN VALUE

On success, 0 is returned. On error, a non-zero error code is returned.

ERRORS

ESRCH
No thread could be found corresponding to that specified by *th*

EINVAL
the thread *th* is already in the detached state

AUTHOR

Xavier Leroy <Xavier.Leroy@inria.fr>

SEE ALSO

pthread_create(3), **pthread_join**(3), **pthread_attr_setdetachstate**(3).

NAME

pthread_self – obtain ID of the calling thread

SYNOPSIS

#include <pthread.h>

pthread_t pthread_self(void);

Compile and link with *-pthread*.

DESCRIPTION

The **pthread_self**() function returns the ID of the calling thread. This is the same value that is returned in **thread* in the **pthread_create**(3) call that created this thread.

RETURN VALUE

This function always succeeds, returning the calling thread's ID.

ERRORS

This function always succeeds.

NOTES

POSIX.1 allows an implementation wide freedom in choosing the type used to represent a thread ID; for example, representation using either an arithmetic type or a structure is permitted. Therefore, variables of type *pthread_t* can't portably be compared using the C equality operator (**==**); use **pthread_equal**(3) instead.

Thread identifiers should be considered opaque: any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results.

Thread IDs are guaranteed to be unique only within a process. A thread ID may be reused after a terminated thread has been joined, or a detached thread has terminated.

The thread ID returned by **pthread_self**() is not the same thing as the kernel thread ID returned by a call to **gettid**(2).

SEE ALSO

pthread_create(3), **pthread_equal**(3), **pthreads**(7)

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***);**

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

lstat(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

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 */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system 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 *st_dev* field describes the device on which this file resides.

The *st_rdev* field describes the device that this file (inode) represents.

The *st_size* field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symlink is the length of the pathname it contains, without a trailing null byte.

The *st_blocks* field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

The *st_blksize* field gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Not all of the Linux file systems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field. (See "noatime" in **mount**(8).)

The field *st_atime* is changed by file accesses, for example, by **execve**(2), **mknod**(2), **pipe**(2), **utime**(2) and **read**(2) (of more than zero bytes). Other routines, like **mmap**(2), may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

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

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	FIFO (named pipe)?
S_ISLNK(m)	symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m)	socket? (Not in POSIX.1-1996.)

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).)

EBADF
fd is bad.

EFAULT
Bad address.

ELOOP
Too many symbolic links encountered while traversing the path.

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).

ENOTDIR
A component of the path is not a directory.

SEE ALSO

access(2), **chmod**(2), **chown**(2), **fstatat**(2), **readlink**(2), **utime**(2), **capabilities**(7), **symlink**(7)

NAME

strcat, strchr, strcmp, strcpy, strdup, strlen, strncat, strncmp, strncpy, strstr, strtok − string operations

SYNOPSIS

#include <string.h>

char *strcat(char ****dest*, const char ****src***);**
Append the string *src* to the string *dest*, returning a pointer *dest*.

char *strchr(const char ****s*, int** *c***);**
Return a pointer to the first occurrence of the character *c* in the string *s*.

int strcmp(const char ****s1*, const char ****s2***);**
Compare the strings *s1* with *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

char *strcpy(char ****dest*, const char ****src***);**
Copy the string *src* to *dest*, returning a pointer to the start of *dest*.

char *strdup(const char ****s*);**
Return a duplicate of the string *s* in memory allocated using **malloc**(3).

size_t strlen(const char ****s*);**
Return the length of the string *s*.

char *strncat(char ****dest*, const char ****src***, size_t** *n***);**
Append at most *n* characters from the string *src* to the string *dest*, returning a pointer to *dest*.

int strncmp(const char ****s1*, const char ****s2***, size_t** *n***);**
Compare at most *n* bytes of the strings *s1* and *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

char *strncpy(char ****dest*, const char ****src***, size_t** *n***);**
Copy at most *n* bytes from string *src* to *dest*, returning a pointer to the start of *dest*.

char *strstr(const char ****haystack*, const char ****needle***);**
Find the first occurrence of the substring *needle* in the string *haystack*, returning a pointer to the found substring.

char *strtok(char ****s*, const char ****delim***);**
Extract tokens from the string *s* that are delimited by one of the bytes in *delim*.

DESCRIPTION

The string functions perform operations on null-terminated strings.

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.

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.

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)