NAME

fclose – close a stream

SYNOPSIS

#include <stdio.h>

int fclose(FILE *_stream_);

DESCRIPTION

The **fclose()** function closes the stream pointed to by *stream* and disassociates it. Any buffers associated with the stream are discarded. If the stream was open for writing, any buffered data is written first.

After a successful **fclose(),** the file descriptor underlying the stream may be reused by subsequent calls to **fdopen**(3) or other functions that allocate file descriptors.

RETURN VALUE

Upon successful completion, **0** is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EBADF

stream is not a valid stream or has been closed.

EIO An I/O error occurred while flushing buffered data.

SEE ALSO

fdopen(3), **fopen**(3), **fscanf**(3)

NAME

fopen, fdopen, fileno – stream open functions

SYNOPSIS

#include <stdio.h>

FILE *fopen(const char *_path_, const char *_mode_);
FILE *fdopen(int _fildes_, const char *_mode_);
int fileno(FILE *_stream_);
int fclose(FILE *_stream_);

DESCRIPTION

The **fopen** function opens the file whose name is the string pointed to by *path* and associates a stream w it.

The argument *mode* points to a string beginning with one of the following sequences (Additional charac may follow these sequences.):

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginn of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. stream is positioned at the beginning of the file.

a Open for appending (writing at end of file). The file is created if it does not exist. The strea positioned at the end of the file.

a+ Open for reading and appending (writing at end of file). The file is created if it does not e The stream is positioned at the end of the file.

The **fdopen** function associates a stream with the existing file descriptor, *fildes*. The *mode* of the str (one of the values "r", "r+", "w", "w+", "a", "a+") must be compatible with the mode of the file descrip The file position indicator of the new stream is set to that belonging to *fildes*, and the error and end-of- indicators are cleared. Modes "w" or "w+" do not cause truncation of the file. The file descriptor is dup'ed, and will be closed when the stream created by **fdopen** is closed. The result of applying **fdopen** shared memory object is undefined.

The function **fileno**() examines the argument *stream* and returns its integer descriptor.

The **fclose**() function flushes the stream pointed to by *stream* (writing any buffered output data us **fflush**(3)) and closes the underlying file descriptor.

RETURN VALUE

Upon successful completion **fopen**, **fdopen** and **freopen** return a **FILE** pointer. Otherwise, **NULL** returned and the global variable *errno* is set to indicate the error. Upon successful completion of **fclos** is returned. Otherwise, **EOF** is returned and *errno* is set to indicate the error.

ERRORS

EINVAL

The *mode* provided to **fopen**, **fdopen**, or **freopen** was invalid.

EBADF

The file descriptor underlying *stream* passed to **fclose** is not valid.

The **fopen**, **fdopen** and **freopen** functions may also fail and set *errno* for any of the errors specified for routine **malloc**(3).

The **fopen** function may also fail and set *errno* for any of the errors specified for the routine **open**(2).

The **fdopen** function may also fail and set *errno* for any of the errors specified for the routine **fcntl**(2).

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 fail
−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 st
ture for the child.

EAGAIN
It was not possible to create a new process because the caller's **RLIMIT_NPROC** resource l
was encountered. To exceed this limit, the process must have either the **CAP_SYS_ADMI**
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 t
and memory required to duplicate the parent's page tables, and to create a unique task structure for
child.

Since version 2.3.3, rather than invoking the kernel's **fork**() system call, the glibc **fork**() wrapper tha
provided as part of the NPTL threading implementation invokes **clone**(2) with flags that provide the sa
effect as the traditional system call. The glibc wrapper invokes any fork handlers that have been establis
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), **cred**
tials(7)

COLOPHON

This page is part of release 3.27 of the Linux *man-pages* project. A description of the project, and infor
tion about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.

NAME

fscanf – formatted input conversion

SYNOPSIS

#include <stdio.h>

int fscanf(FILE ****stream*, const char ****format***, ...);**

DESCRIPTION

The **fscanf()** function reads input from the stream, according to the format string given in *format* (see **sscanf**(3)). Each argument must be a pointer to a variable with a type that is compatible with the corresponding conversion specifier.

On success, the function returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. The return value is **EOF** if an input failure occurs before any conversion.

RETURN VALUE

Returns the number of successfully matched and assigned input fields, or **EOF** if an input failure occurs before any conversion.

ERRORS

EBADF

stream is not a readable stream.

SEE ALSO

scanf(3), **sscanf**(3), **fdopen**(3), **fclose**(3)

NAME

mkfifo, mkfifoat – make a FIFO special file (a named pipe)

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

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

int mkfifo(const char ****pathname*, mode_t** *mode***);**

#include <fcntl.h> /* Definition of AT_* constants */
#include <sys/stat.h>

int mkfifoat(int *dirfd***, const char ****pathname***, mode_t** *mode***);**

DESCRIPTION

mkfifo() makes a FIFO special file with name *pathname*. *mode* specifies the FIFO's permissions. I modified by the process's **umask** in the usual way: the permissions of the created file are (*mode* **~umask**).

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being anonymous communications channel, a FIFO special file is entered into the filesystem by calling **mkfifo**

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in same way as an ordinary file. However, it has to be open at both ends simultaneously before you can p ceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until so other process opens the same FIFO for writing, and vice versa. See **fifo**(7) for nonblocking handling FIFO special files.

mkfifoat()

The **mkfifoat**() function operates in exactly the same way as **mkfifo**(), except for the differences descri here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, a done by **mkfifo**() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value **AT_FDCWD**, then *pathname* is interpreted relativ the current working directory of the calling process (like **mkfifo**()).

If *pathname* is absolute, then *dirfd* is ignored.

See **openat**(2) for an explanation of the need for **mkfifoat**().

RETURN VALUE

On success **mkfifo**() and **mkfifoat**() return 0. On error, −1 is returned and *errno* is set to indicate the err

ERRORS

EACCES

One of the directories in *pathname* did not allow search (execute) permission.

EBADF

(**mkfifoat**()) *pathname* is relative but *dirfd* is neither **AT_FDCWD** nor a valid file descriptor.

EDQUOT

The user's quota of disk blocks or inodes on the filesystem has been exhausted.

EEXIST

pathname already exists. This includes the case where *pathname* is a symbolic link, dangling not.

ENAMETOOLONG

Either the total length of *pathname* is greater than **PATH_MAX**, or an individual filename com nent has a length greater than **NAME_MAX**. In the GNU system, there is no imposed limi

overall filename length, but some filesystems may place limits on the length of a component.

ENOENT
A directory component in *pathname* does not exist or is a dangling symbolic link.

ENOSPC
The directory or filesystem has no room for the new file.

ENOTDIR
A component used as a directory in *pathname* is not, in fact, a directory.

ENOTDIR
(**mkfifoat**()) *pathname* is a relative pathname and *dirfd* is a file descriptor referring to a file other than a directory.

EROFS
pathname refers to a read-only filesystem.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
mkfifo(), **mkfifoat**()	Thread safety	MT-Safe

SEE ALSO

mkfifo(1), **close**(2), **open**(2), **read**(2), **stat**(2), **umask**(2), **write**(2), **fifo**(7)

NAME

unlink – remove directory entry

SYNOPSIS

#include <unistd.h>

int unlink(const char *_path_);

DESCRIPTION

The **unlink**() function removes a link to a file. It removes the link named by the pathname pointed to *path* and decrements the link count of the file referenced by the link.

When the file's link count becomes 0 and no process has the file open, the space occupied by the file will freed and the file will no longer be accessible. If one or more processes have the file open when the link is removed, the link will be removed before **unlink**() returns, but the removal of the file contents be postponed until all references to the file are closed.

RETURN VALUES

Upon successful completion, **0** is returned. Otherwise, **−1** is returned and **errno** is set to indicate the err

ERRORS

The **unlink**() function will fail and not unlink the file if:

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

EACCES Write permission is denied on the directory containing the link to be removed.

ENOENT The named file does not exist or is a null pathname.

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

EPERM The named file is a directory and the effective user of the calling process is not su user.

NAME

wait, waitpid − wait for process to change state

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <sys/wait.h>

pid_t wait(int *_Nullable *wstatus***);**
pid_t waitpid(pid_t *pid***, int *_Nullable** *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.

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 at the time of the call to **waitpid**().

> 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**.

(For Linux-only options, see below.)

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* a ment that the child specified in a call to **exit**(3) or **_exit**(2) or as the argument for a return st ment 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 shoul employed only if **WIFSIGNALED** returned true.

WCOREDUMP(*wstatus***)**
returns true if the child produced a core dump (see **core**(5)). This macro should be employed if **WIFSIGNALED** returned true.

This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementati (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 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 emplo 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 failure, −1 is returned.

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

On failure, each of these calls sets *errno* to indicate the error.

ERRORS

EAGAIN
The PID file descriptor specified in *id* is nonblocking and the process that it refers to has not te nated.

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

ECHILD
(for **waitpid**()) The process specified by *pid* (**waitpid**()) or *idtype* does not exist or is not a chil the calling process. (This can happen for one's own child if the action for **SIGCHLD** is se **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.

ESRCH
(for **wait**() or **waitpid**()) *pid* is equal to **INT_MIN**.

EXAMPLES

The following program demonstrates the use of **fork**(2) and **waitpid**(). The program creates a c process. If no command-line argument is supplied to the program, then the child suspends its execu using **pause**(2), to allow the user to send signals to the child. Otherwise, if a command-line argumen

supplied, then the child exits immediately, using the integer supplied on the command line as the exit status. The parent process executes a loop that monitors the child using **waitpid**(), and uses the W*() macros described above to analyze the wait status value.

The following shell session demonstrates the use of the program:

$ **./a.out &** Child PID is 32360 [1] 32359 $ **kill –STOP 32360** stopped by signal 19 $ **kill –CONT 32360** continued $ **kill –TERM 32360** killed by signal 15 [1]+ Done ./a.out $

Program source

```
#include <stdint.h> #include <stdio.h> #include <stdlib.h> #include <sys/types.h> #include <sys/wait.h>
#include <unistd.h> int main(int argc, char *argv[]) {
   int    wstatus;
   pid_t  cpid, w;
   cpid = fork();
   if (cpid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
   }
   if (cpid == 0) {            /* Code executed by child */
      printf("Child PID is %jd\n", (intmax_t) getpid());
      if (argc == 1)
         pause();                    /* Wait for signals */
      _exit(atoi(argv[1]));
   } else {                    /* Code executed by parent */
      do {
         w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);
         if (w == -1) {
            perror("waitpid");
            exit(EXIT_FAILURE);
         }
         if (WIFEXITED(wstatus)) {
            printf("exited, status=%d\n", WEXITSTATUS(wstatus));
         } else if (WIFSIGNALED(wstatus)) {
            printf("killed by signal %d\n", WTERMSIG(wstatus));
         } else if (WIFSTOPPED(wstatus)) {
            printf("stopped by signal %d\n", WSTOPSIG(wstatus));
         } else if (WIFCONTINUED(wstatus)) {
            printf("continued\n");
         }
      } while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus));
      exit(EXIT_SUCCESS);
   } }
```

SEE ALSO

_exit(2), **clone**(2), **fork**(2), **kill**(2), **ptrace**(2), **sigaction**(2), **signal**(2), **wait4**(2), **pthread_create**(3), **core**(5), **credentials**(7), **signal**(7)