

`close(2)`

Synopsis

```
#include <unistd.h>
```

```
int close(int fd)
```

Description

Close an open descriptor, removing it from the per-process object reference table. If `fd` is the last reference to a particular object, that object will be deactivated.

Normally all open descriptors are closed when a process exits, but since there is an upper limit on the number of descriptors a process may have open at any one time, `close()` is sometimes necessary for programs that deal with many descriptors.

Since processes created with `fork()` inherit all open descriptors from the parent process, it is often necessary to close unwanted or unused descriptors in the child process.

Return Value

`close()` returns:

0 on success.

-1 on failure and sets `errno` to indicate the error.

Errors

`EBADF` `fd` is not an active descriptor.

See Also

`dup(2)`, `execve(2)`, `open(2)`, `pipe(2)`, `socket(2)`

`fork(2)`

Synopsis

```
#include <unistd.h>
```

```
int fork(void)
```

Description

`fork()` creates a new process. The new process (child process) is an exact copy of the calling process except for the following:

- The child process has a unique process ID.
- The child process has a different parent ID (the process ID of the parent process).
- The child process has its own copies of the parent's descriptors. They reference the same underlying objects however, so that for example, pointers in file objects are *shared* by the parent and child processes.
- The child process has its own copy of the parent's open directory streams (see `directory(3)`). The position in the stream is shared by both processes.
- The child's process resource utilisations are set to 0.

Although `fork()` is called once, it returns *twice*, once to the parent (the caller) and once to the newly created child process.

Return Value

On success, `fork()` returns 0 to the child process, and returns the process ID of the child process to the parent process. On failure, `fork()` returns -1, sets `errno` to indicate the error, and does not create a child process.

Errors

EAGAIN The maximum number of processes for the system would be exceeded, or the maximum number of processes under execution by a single user would be exceeded.

ENOMEM There is insufficient swap space for the new process.

See Also

`wait(2)`

`fork(2)`

ipcrm(1)

Synopsis

`ipcrm` [**primitives**]

Description

Remove a message queue, semaphore set, or shared memory ID

`ipcrm` removes one or several messages, semaphores, or shared memory identifiers.

Options

This is an abridged description of the available options. For a more complete description, see the manual page for `ipcrm`:

-q `msgid` removes the message queue identifier `msgid` from the system and destroys the message queue and data structures associated with it.

-m `shmid` removes the shared memory identifier `shmid` from the system. The shared memory segment and data structures associated with it are destroyed after the last detach.

-s `semid` removes the semaphore identifier `semid` from the system and destroys the set of semaphores and data structures associated with it.

See Also

`ipcs(1)`

ipcrm(1)

Synopsis

`ipcs` [**primitives**]

Description

`ipcs` reports status of interprocess communication facilities.

`ipcs` prints information about active interprocess communication facilities, as specified by the primitives provided. If none are specified, information is printed in short form about message queues, shared memory, and semaphores that are currently active in the system.

Options

If any of the primitives **-m**, **-q** or **-s** are specified, information about only the indicated facilities is printed. If none is specified, information about all three is printed.

This is an abridged description of the available options. For a more complete description, see the manual page for `ipcs`:

-m show only information about shared memory.

-q show only information about message queues.

-s show only information about semaphores.

-b print currently allowed size information.

-c print creator's login name and group.

-o print current usage information.

-p print process number information.

-t print time information.

-a all of the above (shorthand for **-b**, **-c**, **-o**, **-p** and **-t**).

See Also

`ipcrm(1)`

ipcs(1)

`getpid(2)`

Synopsis

```
#include <unistd.h>
int  getpid(void)
int  getppid(void)
```

Description

`getpid()` returns the process ID of the calling process.

`getppid()` returns the process ID of the caller's parent process.

See Also

`getpgrp(2)`

`getpid(2)`

`getpgrp(2)`

Synopsis

```
#include <unistd.h>
int  getpgrp(void)
```

Description

`getpgrp()` returns the process group ID of the calling process.

Return Value

`getpgrp()` returns the process group of the indicated process on success. On failure, it returns `-1` and sets `errno` to indicate the error.

Errors

ESRCH There is no process with a process ID equal to `pid`.

See Also

`getpid(2)`, `getppid(2)`

`getpgrp(2)`

man(1)

man(1)

open(2)

Synopsis

```
man [section] title
man -k keyword
```

Description

man displays information from the reference manuals. It can display complete manual pages that you select by title, or one-line summaries selected by keyword (**-k**).

A section, when given, applies to the titles that follow it on the command line (up to the next section, if any). **man** looks in the indicated section of the manual for those titles. section is either a digit (perhaps followed by a single letter indicating the type of manual page), or one of the words **new**, **local**, **old**, or **public**. The abbreviations **n**, **l**, **o** and **p** are also allowed.

If section is omitted, **man** searches all reference sections (giving preference to commands over functions) and prints the first manual page it finds. If no manual page is located, **man** prints an error message.

Options

-k keyword ... print out one-line summaries from the whatis database (table of contents) that contain any of the given keywords.

Examples

List all manual pages containing keyword "write":

```
% man -k write
```

Read the manual page for **write(2)**:

```
% man 2 write
```

Synopsis

```
#include <fcntl.h>

int open(char *path, int flags[, int mode])
```

Description

path is the name of a file. **open()** opens the named file for reading and/or writing, as specified by the flags argument, and returns a descriptor for that file. The flags argument may indicate that the file is to be created if it does not already exist, in which case it is created with the mode specified by mode. (see **chmod(2)**).

Values for flags are constructed by OR'ing flags from the following list. In addition, exactly one of the first three values must be used:

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for both reading and writing.

O_APPEND If this flag is set, the seek pointer will be set to EOF (end of file) prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise the file is created with the mode specified by mode (and modified by the user's umask - see **umask(2)**).

O_TRUNC If the file exists and is successfully opened with **O_RDWR** or **O_WRONLY**, its length is truncated to zero.

Upon successfully opening the file, the seek pointer is positioned to the beginning of the file.

Return Value

open() returns a non-negative file descriptor on success. On failure, it returns -1 and sets **errno** to indicate the error.

Errors

This list is far from complete:

EACCES The required permissions for searching, reading or writing are denied for path.

The file referred to by path does not exist, **O_CREAT** is specified but the user does not have the correct permissions to create the file in the specified directory.

EFAULT path points outside the user's allocated address space.

EISDIR The named file is a directory and the arguments specify that it is to be opened for writing.

EDQUOT The file could not be created because the user's quota of inodes on the system would be exceeded.

ELOOP Too many symbolic links were encountered while translating path.

EMFILE The per-process limit for open file descriptors would be exceeded.

ENAMETOOLONG The length of the path argument exceeds `PATH_MAX` characters.

ENOENT The file does not exist and `O_CREAT` has not been specified.

ENOSPC There is not enough space left to create the file.

See Also

`chmod(2)`, `creat(2)`, `read(2)`, `umask(2)`, `write(2)`

perror(3)

perror(3)

Synopsis

```
#include <stdio.h>
```

```
void perror(char *msg)
```

Description

`perror()` produces a short error message describing the last error encountered during a call to a system or library function. If msg is not `NULL` or an empty string, it is printed before the error message.

The error message printed comes from the current value of `errno`, which is set when errors occur.

See Also

`intro(2)`

`pipe(2)`

Synopsis

```
#include <unistd.h>
```

```
int pipe(int fd[2])
```

Description

`pipe()` creates an I/O mechanism called a pipe and returns two file descriptors, `fd[0]` and `fd[1]`. `fd[0]` is opened for reading and `fd[1]` is opened for writing.

When data is written to `fd[1]` (with `write()`), up to `PIPE_BUF` bytes are buffered before the writing process is blocked. A read only file descriptor `fd[0]` access the data written to `fd[1]` on a first-in first-out basis.

The standard programming model is that after the pipe has been set up, two or more cooperating processes (created by `fork()`) will pass data through the pipe using `read()` and `write()`.

Read calls on an empty pipe (no buffered data) with only one end open (i.e. all write descriptors have been closed) return EOF (end of file).

An error will occur if an attempt is made to write to a pipe with only one end (i.e. all read descriptors have been closed).

Return Value

`pipe()` returns:

0 on success.

-1 on failure and sets `errno` to indicate the error.

Errors

EFAULT The array `fd` is an invalid area of the process's address space.

EMFILE Too many descriptors are active.

ENFILE The system file table is full.

See Also

`fork(2)`, `read(2)`, `write(2)`

`read(2)`

Synopsis

```
#include <unistd.h>
```

```
int read(int fd, char *buf, int nbytes)
```

Description

`read()` attempts to read nbytes bytes of data from the object referenced by `fd` into the buffer pointed to by `buf`.

If nbytes is zero, `read()` takes no action and returns 0.

On objects capable of seeking (such as regular files), `read()` starts at a position given by the current value of the seek pointer associated with `fd`. Upon return from `read()`, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking (such as pipes) are always read from the current position. The value of the pointer associated with such an object is undefined.

On successful completion, `read()` returns the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file which has that many bytes left before EOF (end of file), but in no other case.

If nbyte is not zero and `read()` returns 0, then EOF has been reached.

When attempting to read from an empty pipe that is not marked for non-blocking I/O (this is normally the case), then `read()` will block until data is available or the object has been disconnected. If no process has the pipe open for writing, `read()` returns 0 to indicate end-of-file. A pipe is disconnected when no process has the object open for writing.

Return Value

`read()` returns the number of bytes actually read on success. On failure, it returns -1 and sets `errno` to indicate the error.

Errors

This list is far from complete:

EDABF `fd` is not a valid file descriptor open for reading.

EFAULT `buf` points outside the allocated address space.

EINVAL The seek pointer associated with `fd` was negative.

EIO An error occurred while reading from or writing to the file system.

See Also

`intro(2)`, `lseek(2)`, `open(2)`, `pipe(2)`, `select(2)`, `write(2)`

wait(2)

wait(2)

Synopsis

```
#include <sys/wait.h>
int wait(int *status)
int waitpid(int pid, int *status, int options)
```

Description

`wait()` and `waitpid()` are both used to obtain exit status from a terminated child process, as well as to remove process resource usage information from the process table. It is *always* necessary for a process to call one of the `wait()` functions for each of its child processes, in order to avoid zombie processes.

`wait()` waits for status from *any* of the caller's terminated child processes. If any child has terminated, `wait()` updates `status` with the exit status of the child and returns the child's process ID to the caller. If there are no children, `wait()` returns a value of `-1`. If there are only running or stopped children, the calling process is blocked until a child process terminates. If `status` is `NULL`, `wait()` behaves as described but without updating the status.

`waitpid()` behaves identically to `wait()` if `pid` is `-1` and `options` is zero. Otherwise, the behaviour of `waitpid()` is modified by `pid` and `options` as follows:

- `pid` specifies for which processes status is requested. If `pid` is greater than zero, it specifies the process ID of a single child process for which status is requested. If `pid` is `-1`, status is requested for *any* child process and `waitpid()` behaves like `wait()`.
- `options` are one or a combination of `WNOHANG` and `WUNTRACED`. `WNOHANG` prevents `waitpid()` from blocking the calling process when there is no data immediately available. `WUNTRACED` causes `waitpid()` to report the status of stopped processes.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are assigned the parent process ID of 1 (see `init(8)`).

There are several more system calls in the `wait()` family that offer alternate interfaces.

Return Value

If `wait()` or `waitpid()` returns due to a stopped or terminated child process, the process ID of the child is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

If `waitpid()` was used with `WNOHANG`, `pid` specifies at least one child process for which status is not available, and status is not available for any of the processes specified by `pid`, zero is returned. Otherwise `-1` is returned and `errno` is set to indicate the error.

Errors

`wait()` may fail and return one of the following:

ECHILD The caller has no existing and not-yet-waited-for child processes.

EFAULT `status` points to an illegal address.

`waitpid()` may fail and return one of the following:

ECHILD The process or processes specified by `pid` do not exist or are not children of the caller.

EINVAL The value of `options` is invalid.

See Also

`exit(2)`, `fork(2)`, `pause(3)`

`write(2)`

`write(2)`

See Also

`lseek(2)`, `open(2)`, `pipe(2)`

Synopsis

```
#include <unistd.h>

int write(int fd, char *buf, int nbytes)
```

Description

`write()` attempts to write nbytes bytes of data from the buffer buf to the object referenced by the descriptor fd.

If nbytes is zero, `write()` takes no action and returns 0.

On objects capable of seeking (such as regular files), `write()` starts at a position given by the current value of the seek pointer associated with fd. Upon return from `write()`, the pointer is incremented by the number of bytes actually written.

Objects that are not capable of seeking (such as pipes) are always written from the current position. The value of the pointer associated with such an object is undefined.

If the object was opened with the `O_APPEND` flag, the seek pointer is set to the end of the file prior to each write.

Return Value

On success, `write()` returns the number of bytes actually written. On failure, it returns -1 and sets `errno` to indicate the error.

Errors

`write()` will fail and the seek pointer will remain unchanged if one or more of the following are true (this list is not complete):

EBADF fd is not a valid file descriptor opened for writing.

EDQUOT The user's disk quota would be exceeded.

EFAULT Part or all of the data to be written extends outside the processes allocated address space.

EFBIG An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.

EIO An error occurred while reading from or writing to the file system.

ENOSPC There is no free space on the file system.

EPIPE An attempt was made to write to a pipe that was not open for reading by any process. Note that an attempted write of this kind also causes you to receive a `SIGPIPE` signal from the kernel. If you've not made a special provision to catch or ignore this signal, then your process dies.