close(2) close(2)

NAME

close − close a file descriptor

SYNOPSIS

#include <unistd.h>

int close(int *fd***);**

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused.

RETURN VALUE

close() returns zero on success. On error, −1 is returned, and *errno* is set appropriately.

---

closedir(3) closedir(3)

NAME

closedir − close a directory

SYNOPSIS

#include <sys/types.h>

#include <dirent.h>

int closedir(DIR ****dirp*);**

DESCRIPTION

The **closedir**() function closes the directory stream associated with *dirp*.

RETURN VALUE

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

---

dup(2) dup(2)

NAME

dup, dup2 − duplicate a file descriptor

SYNOPSIS

#include <unistd.h>

int dup(int *oldfd***);**
int dup2(int *oldfd***, int** *newfd***);**

DESCRIPTION

The **dup**() system call creates a copy of the file descriptor *oldfd*, using the lowest-numbered unused file descriptor for the new descriptor.

dup2()

The **dup2**() system call performs the same task as **dup**(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in *newfd*. If the file descriptor *newfd* was previously open, it is silently closed before being reused.

The steps of closing and reusing the file descriptor *newfd* are performed *atomically*.

RETURN VALUE

On success, these system calls return the new file descriptor. On error, −1 is returned, and *errno* is set appropriately.

---

exec(2) exec(2)

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 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[]);

where *argc* is the argument count, and *argv* is an array of character pointers to the arguments themselves. 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 *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.

---

fnmatch(3) fnmatch(3)

NAME

fnmatch − match filename or pathname

SYNOPSIS

#include <fnmatch.h>

int fnmatch(const char ****pattern*, const char ****string***, int** *flags***);**

DESCRIPTION

The **fnmatch**() function checks whether the *string* argument matches the *pattern* argument, which is a shell wildcard pattern.

The *flags* argument modifies the behavior; it is the bitwise OR of zero or more flags.

RETURN VALUE

Zero if *string* matches *pattern*, **FNM_NOMATCH** if there is no match or another nonzero value if there is an error.

---

fork(2) fork(2)

NAME

fork – create a child process

SYNOPSIS

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

pid_t fork(void);

DESCRIPTION

fork() creates a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process is an exact duplicate of the parent process except for the following points:

* The child has its own unique process ID.

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

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.

open(2) open(2)

NAME

open, creat – open and possibly create a file

SYNOPSIS

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

int open(const char ***_pathname_, int** _flags_**);**
int open(const char ***_pathname_, int** _flags_**, mode_t** _mode_**);**

int creat(const char ***_pathname_, mode_t** _mode_**);**

DESCRIPTION

The **open**() system call opens the file specified by *pathname*. If the specified file does not exist, it may optionally (if **O_CREAT** is specified in *flags*) be created by **open**().

The return value of **open**() is a file descriptor.

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

In addition, zero or more flags can be bitwise-*or*'d in *flags*. The *file creation flags* are

O_APPEND

The file is opened in append mode.

O_CREAT

If *pathname* does not exist, create it as a regular file.

The owner (user ID) of the new file is set to the effective user ID of the process.

The *mode* argument specifies the file mode bits be applied when a new file is created. This argument must be supplied when **O_CREAT** is specified in *flags*; otherwise *mode* is ignored.

creat()

A call to **creat**() is equivalent to calling **open**() with *flags* equal to **O_CREAT|O_WRONLY|O_TRUNC**.

RETURN VALUE

open(), **openat**(), and **creat**() return the new file descriptor, or −1 if an error occurred (in which case, *errno* is set appropriately).

opendir(3) opendir(3)

NAME

opendir, fdopendir – open a directory

SYNOPSIS

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

DIR *opendir(const char ***_name_);**
DIR *fdopendir(int _fd_**);**

DESCRIPTION

The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream.

After a successful call to **fdopendir**(), *fd* is used internally by the implementation, and should not otherwise be used by the application.

RETURN VALUE

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

pipe(2) pipe(2)

NAME

pipe, pipe2 – create pipe

SYNOPSIS

#include <unistd.h>

int pipe(int _pipefd_**[2]);**

DESCRIPTION

pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe.

RETURN VALUE

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

read(2) read(2)

NAME

read – read from a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t read(int _fd_**, void ***_buf_**, size_t** _count_**);**

DESCRIPTION

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

If *count* is zero, **read**() *may* detect the errors described below. In the absence of any errors, or if **read**() does not check for errors, a **read**() with a *count* of 0 returns zero and has no other effects.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested.

On error, −1 is returned, and *errno* is set appropriately. In this case, it is left unspecified whether the file position (if any) changes.