chdir(2) chdir(2)

NAME

chdir, fchdir – change working directory

SYNOPSIS

#include <unistd.h>

int chdir(const char ****path*);**
int fchdir(int *fd***);**

DESCRIPTION

chdir() changes the current working directory of the calling process to the directory specified in *path*.
fchdir() is identical to **chdir**(); the only difference is that the directory is given as an open file descriptor.

RETURN VALUE

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

---

chmod(2) chmod(2)

NAME

chmod, fchmod, fchmodat – change permissions of a file

SYNOPSIS

#include <sys/stat.h>

int chmod(const char ****pathname*, mode_t** *mode***);**
int fchmod(int *fd***, mode_t** *mode***);**

DESCRIPTION

The **chmod**() and **fchmod**() system calls change a files mode bits. **fchmod**() is identical to **chmod**(); but the directory is given as an open file descriptor. The new file mode is specified in *mode*.

RETURN VALUE

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

---

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 <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 duplicates a file descriptor.
The **dup2**() system call replaces the file descriptor specified in *newfd* with a copy of *oldfd*. 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: 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.

---

getline(3) getline(3)

NAME

getline, getdelim − delimited string input

SYNOPSIS

#include <stdio.h>

ssize_t getline(char *****lineptr*, size_t ****n***, FILE ****stream***);**
ssize_t getdelim(char *****lineptr*, size_t ****n***, int** *delim***, FILE ****stream***);**

DESCRIPTION

getline() reads an entire line from *stream*, storing the address of the buffer containing the text into **lineptr*. The buffer is null-terminated and includes the newline character, if one was found.

If **lineptr* is set to NULL and **n* is set 0 before the call, then **getline**() will allocate a buffer for storing the line. This buffer should be freed by the user program even if **getline**() failed.

Alternatively, before calling **getline**(), **lineptr* can contain a pointer to a **malloc**(3)−allocated buffer **n* bytes in size. If nessecary, **getline**() will resizes it.

getdelim() works like **getline**(), except that a line delimiter is specified via the *delimiter* argument.

RETURN VALUE

On success, the number of characters read, including the delimiter, but excluding the terminating null byte. On error and end-of-file −1 is returned. In the event of an error, *errno* is set appropriately.

---

kill(2) kill(2)

NAME

kill − send signal to a process

SYNOPSIS

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

int kill(pid_t *pid***, int** *sig***);**

DESCRIPTION

The **kill**() system call can be used to send any signal to any process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*. If *pid* equals −1, then *sig* is sent to every process for which the calling process has permission to send signals.

If *sig* is 0, then no signal is sent, but existence and permission checks are still performed.

RETURN VALUE

On success (at least one signal was sent), zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EINVAL An invalid signal was specified.
EPERM The process does not have permission to send the signal to any of the target processes.
ESRCH The process or process group does not exist.

---

lseek(2) lseek(2)

NAME

lseek − reposition read/write file offset

SYNOPSIS

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

off_t lseek(int *fd***, off_t** *offset***, int** *whence***);**

DESCRIPTION

lseek() repositions the file offset of the open file description associated with the file descriptor *fd* to the argument *offset* according to the directive *whence* as follows:
SEEK_SET The file offset is set to *offset* bytes.
SEEK_CUR The file offset is set to its current location plus *offset* bytes.
SEEK_END The file offset is set to the size of the file plus *offset* bytes.

lseek() allows the file offset to be set beyond the end of the file.

RETURN VALUE

Upon successful completion, **lseek**() returns the resulting offset location as measured in bytes from the beginning of the file. On error, the value *(off_t) −1* is returned and *errno* is set to indicate the error.

malloc(3) malloc(3)

NAME

malloc, free, calloc, realloc − allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t *size***);**
void free(void **ptr***);**
void *calloc(size_t *nmemb***, size_t** *size***);**
void *realloc(void **ptr***, size_t** *size***);**

DESCRIPTION

The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory.

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**(), or **realloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

The **calloc**() function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. Added memory will *not* be initialized. If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*; if *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)*. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc**(), **calloc**(), or **realloc**().

RETURN VALUE

The **malloc**() and **calloc**() functions return a pointer to the allocated memory. On error, these functions return NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero, or by a successful call to **calloc**() with *nmemb* or *size* equal to zero.

The **realloc**() function returns a pointer to the newly allocated memory, or NULL if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to **free**() is returned. If **realloc**() fails, the original block is left untouched; it is not freed or moved.

---

open(2) open(2)

NAME

open, creat − open and possibly create a file

SYNOPSIS

#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 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 of the following flags can be bitwise-*or*'d in *flags*:
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.

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

RETURN VALUE

on success: return the new file descriptor; on error:return −1, 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.

---

perror(3) perror(3)

NAME

perror − print a system error message

SYNOPSIS

#include <stdio.h>

void perror(const char ***_s_);**

#include <errno.h>

int _errno_**;**

DESCRIPTION

The **perror**() function produces a message on standard error describing the last error encountered during a call to a system or library function.

First (if *s* is not NULL and **s* is not a null byte ('\0')), the argument string *s* is printed, followed by a colon and a blank. Then an error message corresponding to the current value of *errno* and a new-line.

When a system call fails, it usually returns −1 and sets the variable *errno* to a value describing what went wrong.

---

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.

pthread_create(3) pthread_create(3)

NAME

pthread_create – create a new thread

SYNOPSIS

#include <pthread.h>

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

DESCRIPTION

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

The new thread terminates in one of the following ways: It calls **pthread_exit**(3); it returns from *start_routine*(); it is canceled via **pthread_cancel**(3); any of the threads in the process calls **exit**(3); or the main thread performs a return from *main*().

The *attr* argument points to a *pthread_attr_t* structure containing startup attributes. If *attr* is NULL, then default attributes are used.

RETURN VALUE

On success, **pthread_create**() returns 0 and the thread ID is stored in **thread*; on error, it returns an error number, and the contents of **thread* are undefined.

---

pthread_detach(3) pthread_detach(3)

NAME

pthread_detach – detach a thread

SYNOPSIS

#include <pthread.h>

int pthread_detach(pthread_t *thread***);**

DESCRIPTION

The **pthread_detach**() function marks the thread identified by *thread* as detached. Resources of a detached thread are automatically released on termination.

Attempting to detach an already detached thread results in unspecified behavior.

RETURN VALUE

On success, **pthread_detach**() returns 0; on error, it returns an error number.

---

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.

---

readdir(3) readdir(3)

NAME

readdir – read a directory

SYNOPSIS

#include <dirent.h>

struct dirent *readdir(DIR ****dirp*);**

DESCRIPTION

The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*. It returns NULL on reaching the end of the directory stream or if an error occurred.

The *dirent* structure is defined as follows:

```
struct dirent {
    long            d_ino;      /* inode number */
    off_t           d_off;      /* offset to the next dirent */
    unsigned short  d_reclen;   /* length of this record */
    unsigned char   d_type;     /* type of file; not supported by all filesystem types */
    char            d_name[256]; /* filename */
};
```

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure.

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.

---

sem_getvalue(3) sem_getvalue(3)

NAME

sem_getvalue – get the value of a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_getvalue(sem_t ****sem*, int ****sval***);**

DESCRIPTION

sem_getvalue() places the current value of the semaphore pointed to *sem* into the integer pointed to by *sval*.

RETURN VALUE

sem_getvalue() returns 0 on success; on error, −1 is returned and *errno* is set appropriately.

---

sem_init(3) sem_init(3)

NAME

sem_init – initialize an unnamed semaphore

SYNOPSIS

#include <semaphore.h>

int sem_init(sem_t ****sem*, int** *pshared***, unsigned int** *value***);**

DESCRIPTION

sem_init() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore.

The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process (0), or between processes (1). Initializing a semaphore that has already been initialized results in undefined behavior.

RETURN VALUE

sem_init() returns 0 on success; on error, −1 is returned, and *errno* is set appropriately.

NAME

sem_post – unlock a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_post(sem_t ****sem*);**

DESCRIPTION

sem_post() increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a **sem_wait**(3) call will be woken up and proceed to lock the semaphore.

RETURN VALUE

sem_post() returns 0 on success; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

NAME

sem_wait, sem_timedwait – lock a semaphore

SYNOPSIS

#include <semaphore.h>

int sem_wait(sem_t ****sem*);**
int sem_trywait(sem_t ****sem*);**

DESCRIPTION

sem_wait() decrements (locks) the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

sem_trywait() is the same as **sem_wait**(), except that if the decrement cannot be immediately performed, then call returns an error (*errno* set to **EAGAIN**) instead of blocking.

RETURN VALUE

on success: 0; on error, the value of the semaphore is left unchanged, −1 is returned, and *errno* is set to indicate the error.

ERRORS

EINTR The call was interrupted by a signal handler
EINVAL *sem* is not a valid semaphore.
EAGAIN The operation could not be performed without blocking (sem_trywait() only).

NAME

sigaction – examine and change a signal action

SYNOPSIS

#include <signal.h>

int sigaction(int *signum***, const struct sigaction ****act***,**
struct sigaction ****oldact*);**

DESCRIPTION

The **sigaction**() system call is used to change the action taken by a process on receipt of the signal *signum*.

If *act* is non-NULL, the new action for signal *signum* is installed from *act*. If *oldact* is non-NULL, the previous action is saved in *oldact*.

The *sigaction* structure is defined as follows:

```
struct sigaction {
    void        (*sa_handler)(int);
    ...
};
```

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.

RETURN VALUE

sigaction() returns 0 on success; on error, −1 is returned, and *errno* is appropriately

NAME

stat, fstat, lstat – get file status

SYNOPSIS

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

int stat(const char ****pathname*, struct stat ****statbuf***);**
int fstat(int *fd***, struct stat ****statbuf***);**
int lstat(const char ****pathname*, struct stat ****statbuf***);**

DESCRIPTION

These functions return information about a file, in the buffer pointed to by *statbuf*. **stat**() retrieves information about the file pointed to by *pathname*; **lstat**() is identical to **stat**(), except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that it refers to. **fstat**() is identical to **stat**(), except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

```
struct stat {
    ino_t           st_ino;     /* Inode number */
    mode_t          st_mode;    /* File type and mode */
    nlink_t         st_nlink;   /* Number of hard links */
    uid_t           st_uid;     /* User ID of owner */
    gid_t           st_gid;     /* Group ID of owner */
    off_t           st_size;    /* Total size, in bytes */
    struct timespec st_atim;    /* Time of last access */
    struct timespec st_mtim;    /* Time of last modification */
    struct timespec st_ctim;    /* Time of last status change */
};
```

RETURN VALUE

On success: 0; on error, −1 is returned, and *errno* is set appropriately.

NAME

strtok − extract tokens from strings

SYNOPSIS

#include <string.h>

char *strtok(char *_str_, const char *_delim_);

DESCRIPTION

The **strtok**() function breaks a string into a sequence of zero or more nonempty tokens. On the first call to **strtok**(), the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* must be NULL.

The *delim* argument specifies a set of bytes that delimit the tokens in the parsed string. The caller may specify different strings in *delim* in successive calls that parse the same string.

RETURN VALUE

strtok() returns a pointer to the next token, or NULL if there are no more tokens.

NAME

strtol − convert a string to a long integer

SYNOPSIS

#include <stdlib.h>

long int strtol(const char *_nptr_, char **_endptr_, int _base_);

DESCRIPTION

The **strtol**() function converts the initial part of the string in *nptr* to a long integer value according to the given *base*, which must be between 2 and 36 inclusive, or be the special value 0.

The string may begin with an arbitrary amount of white space (as determined by **isspace**(3)) followed by a single optional '+' or '−' sign. If *base* is zero or 16, the string may then include a "0x" or "0X" prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is '0', in which case it is taken as 8 (octal).

The remainder of the string is converted to a *long int* value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter 'A' in either uppercase or lowercase represents 10, 'B' represents 11, and so forth, with 'Z' representing 35.)

If *endptr* is not NULL, **strtol**() stores the address of the first invalid character in **endptr*. If there were no digits at all, **strtol**() stores the original value of *nptr* in **endptr* (and returns 0). In particular, if **nptr* is not '\0' but **endptr* is '\0' on return, the entire string is valid.

RETURN VALUE

The **strtol**() function returns the result of the conversion, unless the value would underflow or overflow, returning LONG_MIN or LONG_MAX and settting errno to ERANGE.

NAME

wait, waitpid − wait for process to change state

SYNOPSIS

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

pid_t wait(int *_wstatus_);

pid_t waitpid(pid_t *pid*, **int *_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. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child.

wait() and waitpid()

The **wait**() and **waitpid()** system calls suspends execution of the calling thread until one of its (specified) children terminates. The call *wait(&wstatus)* is equivalent to waitpid(−1, &wstatus, 0);

The value of *pid* can be:
−1 meaning wait for any child process.
> 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.
WCONTINUED also return if a stopped child has been resumed.

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:
WIFEXITED(*wstatus*) returns true if the child terminated normally
WEXITSTATUS(*wstatus*) returns the exit status of the child. Use only if **WIFEXITED** returned true.

RETURN VALUE

On success: child pid; on error: -1 and errno is set appropriately; if **WNOHANG** is specified and children exist but did not exit: 0

NAME

write − write to a file descriptor

SYNOPSIS

#include <unistd.h>

ssize_t write(int *fd*, **const void *_buf_, size_t** *count*);

DESCRIPTION

write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

RETURN VALUE

On success, the number of bytes written is returned (zero indicates nothing was written). 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.
