NAME

alarm − set an alarm clock for delivery of a signal

SYNOPSIS

#include <unistd.h>

unsigned int alarm(unsigned int *seconds***);**

DESCRIPTION

alarm() arranges for a **SIGALRM** signal to be delivered to the calling process in *seconds* seconds.

If *seconds* is zero, no new **alarm**() is scheduled.

In any event any previously set **alarm**() is canceled.

RETURN VALUE

alarm() returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

CONFORMING TO

SVr4, POSIX.1-2001, 4.3BSD.

NAME

opendir − open a directory / readdir − read a directory

SYNOPSIS

#include <sys/types.h>

#include <dirent.h>

DIR *opendir(const char ***name*);**

struct dirent *readdir(DIR ***dir*);**
int readdir_r(DIR ***dirp*, struct dirent ***entry***, struct dirent ****result***);**

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 or NULL if an error occurred.

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.

DESCRIPTION readdir_r

The **readdir_r()** function initializes the structure referenced by *entry* and stores a pointer to this structure in *result*. On successful return, the pointer returned at **result* will have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer will have the value NULL.

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 */
        off_t           d_off;                  /* offset to the next dirent */
        unsigned short  d_reclen;               /* length of this record */
        unsigned char   d_type;         /* type of file */
        char            d_name[256];    /* filename */
};
```

RETURN VALUE

The **readdir()** function returns a pointer to a dirent structure, or NULL if an error occurs or end-of-file is reached.

readdir_r() returns 0 if successful or an error number to indicate failure.

ERRORS

EACCES
Permission denied.

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

ENOTDIR
name is not a directory.

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

gets, fgets − get a string from a stream
fputs, puts − output of strings

SYNOPSIS

```
#include <stdio.h>

char *gets(char *s);

char *fgets(char *s, int n, FILE *stream);

int fputs(const char *s, FILE *stream);

int puts(const char *s);
```

DESCRIPTION gets/fgets

The **gets( )** function reads characters from the standard input stream (see **intro**(3)), **stdin**, into the array pointed to by *s*, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

The **fgets( )** function reads characters from the *stream* into the array pointed to by *s*, until *n*−1 characters are read, or a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

When using **gets( )**, if the length of an input line exceeds the size of *s*, indeterminate behavior may result. For this reason, it is strongly recommended that **gets( )** be avoided in favor of **fgets( )**.

RETURN VALUES

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a null pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a null pointer is returned and the error indicator for the stream is set. If end-of-file is encountered, the **EOF** indicator for the stream is set. Otherwise *s* is returned.

ERRORS

The **gets( )** and **fgets( )** functions will fail if data needs to be read and:

EOVERFLOW The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding *stream*.

DESCRIPTION puts/fputs

fputs() writes the string *s* to *stream*, without its trailing **’\0’**.

puts() writes the string *s* and a trailing newline to *stdout*.

Calls to the functions described here can be mixed with each other and with calls to other output functions from the **stdio** library for the same output stream.

RETURN VALUE

puts() and **fputs()** return a non - negative number on success, or **EOF** on error.

NAME

kill − send signal to a process

SYNOPSIS

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

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

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

kill(): _POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _POSIX_SOURCE

DESCRIPTION

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

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.

If *pid* equals −1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.

If *pid* is less than −1, then *sig* is sent to every process in the process group whose ID is −*pid*.

If *sig* is 0, then no signal is sent, but error checking is still performed; this can be used to check for the existence of a process ID or process group ID.

For a process to have permission to send a signal it must either be privileged (under Linux: have the **CAP_KILL** capability), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of **SIGCONT** it suffices when the sending and receiving processes belong to the same session.

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 pid or process group does not exist. Note that an existing process might be a zombie, a process which already committed termination, but has not yet been **wait**(2)ed for.

NAME

sigaction − POSIX signal handling functions.

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 a specific signal.

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

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 something like

```
struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask;
   int sa_flags;
   void (*sa_restorer)(void);
}
```

On some architectures a union is involved - do not assign to both *sa_handler* and *sa_sigaction*.

The *sa_restorer* element is obsolete and should not be used. POSIX does not specify a *sa_restorer* element.

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.

sa_mask gives a mask of signals which should be blocked during execution of the signal handler. In addition, the signal which triggered the handler will be blocked, unless the **SA_NODEFER** or **SA_NOMASK** flags are used.

sa_flags specifies a set of flags which modify the behaviour of the signal handling process. It is formed by the bitwise OR of zero or more of the following:

SA_NOCLDSTOP
If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when child processes receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN** or **SIGTTOU**).

SA_RESTART
Provide behaviour compatible with BSD signal semantics by making certain system calls restartable across signals.

RETURN VALUES

sigaction returns 0 on success and -1 on error.

ERRORS

EINVAL
An invalid signal was specified. This will also be generated if an attempt is made to change the action for **SIGKILL** or **SIGSTOP**, which cannot be caught.

SEE ALSO

kill(1), **kill**(2), **killpg**(2), **pause**(2), **sigsetops**(3),

NAME

sigprocmask − change and/or examine caller’s signal mask
sigsuspend − install a signal mask and suspend caller until signal

SYNOPSIS

#include <signal.h>

int sigprocmask(int *how***, const sigset_t ***set***, sigset_t ***oset***);**

int sigsuspend(const sigset_t ***set*);**

DESCRIPTION sigprocmask

The **sigprocmask( )** function is used to examine and/or change the caller’s signal mask. If the value is **SIG_BLOCK**, the set pointed to by the argument *set* is added to the current signal mask. If the value is **SIG_UNBLOCK**, the set pointed by the argument *set* is removed from the current signal mask. If the value is **SIG_SETMASK**, the current signal mask is replaced by the set pointed to by the argument *set*. If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is NULL, the value *how* is not significant and the caller’s signal mask is unchanged; thus, the call can be used to inquire about currently blocked signals.

If there are any pending unblocked signals after the call to **sigprocmask( )**, at least one of those signals will be delivered before the call to **sigprocmask( )** returns.

It is not possible to block those signals that cannot be ignored this restriction is silently imposed by the system. See **sigaction**(2).

If **sigprocmask( )** fails, the caller’s signal mask is not changed.

RETURN VALUES

On success, **sigprocmask( )** returns **0**. On failure, it returns **−1** and sets **errno** to indicate the error.

ERRORS

sigprocmask( ) fails if any of the following is true:

EFAULT *set* or *oset* points to an illegal address.

EINVAL The value of the *how* argument is not equal to one of the defined values.

DESCRIPTION sigsuspend

sigsuspend( ) replaces the caller’s signal mask with the set of signals pointed to by the argument *set* and then suspends the caller until delivery of a signal whose action is either to execute a signal catching function or to terminate the process.

If the action is to terminate the process, **sigsuspend( )** does not return. If the action is to execute a signal catching function, **sigsuspend( )** returns after the signal catching function returns. On return, the signal mask is restored to the set that existed before the call to **sigsuspend( )**.

It is not possible to block those signals that cannot be ignored (see **signal**(5)); this restriction is silently imposed by the system.

RETURN VALUES

Since **sigsuspend( )** suspends process execution indefinitely, there is no successful completion return value. On failure, it returns −1 and sets **errno** to indicate the error.

ERRORS

sigsuspend( ) fails if either of the following is true:

EFAULT *set* points to an illegal address.

EINTR A signal is caught by the calling process and control is returned from the signal catching function.

SEE ALSO

sigaction(2), **sigsetops**(3C),

NAME

sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember − manipulate sets of signals

SYNOPSIS

#include <signal.h>

int sigemptyset(sigset_t ***_set_);**

int sigfillset(sigset_t ***_set_);**

int sigaddset(sigset_t ***_set_, int** _signo_**);**

int sigdelset(sigset_t ***_set_, int** _signo_**);**

int sigismember(sigset_t ***_set_, int** _signo_**);**

DESCRIPTION

These functions manipulate _sigset_t_ data types, representing the set of signals supported by the implementation.

sigemptyset( ) initializes the set pointed to by _set_ to exclude all signals defined by the system.

sigfillset( ) initializes the set pointed to by _set_ to include all signals defined by the system.

sigaddset( ) adds the individual signal specified by the value of _signo_ to the set pointed to by _set_.

sigdelset( ) deletes the individual signal specified by the value of _signo_ from the set pointed to by _set_.

sigismember( ) checks whether the signal specified by the value of _signo_ is a member of the set pointed to by _set_.

Any object of type _sigset_t_ must be initialized by applying either **sigemptyset( )** or **sigfillset( )** before applying any other operation.

RETURN VALUES

Upon successful completion, the **sigismember( )** function returns a value of one if the specified signal is a member of the specified set, or a value of 0 if it is not. Upon successful completion, the other functions return a value of 0. Otherwise a value of −1 is returned and **errno** is set to indicate the error.

ERRORS

sigaddset( ), **sigdelset( )**, and **sigismember( )** will fail if the following is true:

EINVAL The value of the _signo_ argument is not a valid signal number.

sigfillset( ) will fail if the following is true:

EFAULT The _set_ argument specifies an invalid address.

SEE ALSO

sigaction(2), **sigpending**(2), **sigprocmask**(2), **sigsuspend**(2), **attributes**(5), **signal**(5)

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

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.

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), **wstat**(5)