LynxOS Reference Manual

LynxOS Version 2.5
DOC-0171

System Calls

Product names mentioned in the *LynxOS Reference Manuals* are trademarks of their respective manufacturers and are used here only for identification purposes.

U.S. Patents 5,469,571; 5,594,903

Printed in the United States of America.

Lynx Real-Time Systems, Inc. makes no representations, express or implied, with respect to this documentation or the software it describes, including (with no limitation) any implied warranties of utility or fitness for any particular purpose; all such warranties are expressly disclaimed. Neither Lynx Real-Time Systems, Inc., nor its distributors, nor its dealers shall be liable for any indirect, incidental, or consequential damages under any circumstances.

(The exclusion of implied warranties may not apply in all cases under some statutes, and thus the above exclusion may not apply. This warranty provides the purchaser with specific legal rights. There may be other purchaser rights which vary from state to state within the United States of America.)

Contents

System Calls

System Calls

NAME

`Intro` – LynxOS system calls

DESCRIPTION

LynxOS system calls are all available directly to any C program. They are implemented as ordinary function calls to "glue" routines in the system library which trap to the OS code.

Note that many system calls use data structures which should be obtained in a program from appropriate header files. Necessary header files are listed with the system call synopsis. If a call returns some type other than `int`, it must be explicitly declared by the calling module.

NOTE

Examples referred to under **SEE ALSO** can be found on-line in the directory `/src/examples`.

NAME

`acancel` – try to cancel an asynchronous request

SYNOPSIS

```
#include <sys/aio.h>

int acancel(fd, aiocbp)
int fd;
struct aiocb *aiocbp;
```

DESCRIPTION

`acancel` attempts to cancel the asynchronous request(s) submitted for file **fd**.

If **aiocbp** is NULL, all asynchronous operations pending against **fd** are cancelled. If **aiocbp** is non-NULL, the particular request identified by **aiocbp** is cancelled. Note that if the **aiocbp** is non-NULL, and the **aiocbp** was re-used, the last request which the **aiocbp** was used for becomes the request to be cancelled.

If the asynchronous operation is successfully cancelled:

1. The associated **aio_errno** field is set to `ECANCELED` and the **aio_nobytes** is set to -1.
2. Any event which is set to be sent on the completion of the operation will not be sent.
3. An `iosuspend` call for a cancelled asynchronous operation will consider the operation to be completed.

Some asynchronous operations cannot be cancelled. `acancel` will fail if the operation has either already completed, or has advanced to such a stage that it cannot be cancelled. If the operation cannot be cancelled, it will complete normally, and the associated **aiocbp** is not changed.

NOTE

If **aiocbp** is not NULL, and the **aiocbp** was NOT last used for the specific operation to be cancelled, undesirable results may occur (such as the operation which used **aiocbp** most recently being cancelled). Currently LynxOS I/O operations cannot be canceled.

DIAGNOSTICS

If `acancel` succeeds it returns one of 3 possible return values:

`AIO_CANCELED` The requested operations were all successfully cancelled.

`AIO_ALLDONE` The requested operations have all already completed.

`AIO_NOTCANCELED` At least one of the requested operations could not be cancelled.

Note: If one is attempting to cancel all operations on a given file **fd**, `AIO_NOTCANCELED` may be returned although some of the outstanding operations were cancelled.

If `acancel` fails, -1 is returned, and **errno** will contain one of the following:

`EBADF` **fd** is not a valid file descriptor. **aiocbp** is not NULL and **fd** does not correspond with the operation referenced by **aiocbp**. This may happen if **aiocbp** was re-used.

`EFAULT` **aiocbp** is not a valid address.

`EINVAL` **aiocbp** was provided but is invalid. This may happen if the request is cancelled twice.

PORTABILITY

A POSIX-conforming application should not rely on `acancel` working, as it is implementation-defined, whether an operation can be cancelled or not.

CONFORMANCE

`acancel` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Library Functions - `iosuspend(), listio(), aread(), awrite()`

The POSIX asynchronous I/O calls are:

System Calls - `acancel()`

Library Functions – `iosuspend(), aread(), awrite(), listio(), afsync()`

Example - `acancel.c`

NAME

`accept` – accept a connection on a socket

SYNOPSIS

```
#include <types.h>
#include <socket.h>

ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr *addr;
int *addrlen;
```

DESCRIPTION

The argument **s** is a socket which has been created with `socket`, bound to an address with `bind`, and is listening for connections after a `listen`. `accept` extracts the first connection on the queue of pending connections, creates a new socket with the same properties of **s** and allocates a new file descriptor, **ns**, for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, `accept` blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, `accept` returns an error as described below. The accepted socket, **ns**, may not be used to accept more connections. The original socket **s** remains open.

The argument **addr** is a result parameter which is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the **addr** parameter is determined by the domain in which the communication is occurring. The **addrlen** is a value-result parameter; it should initially contain the amount of space pointed to by **addr**; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with `SOCK_STREAM`.

DIAGNOSTICS

`accept` returns a non-negative integer which is a descriptor for the accepted socket if successful, or `-1` if not.

If the call fails, **errno** will contain one of the following:

EBADF	The descriptor is invalid.
ENOTSOCK	The descriptor references a file, not a socket.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EFAULT	The **addr** parameter is not in a writable part of the user address space.
EWOULDBLOCK	Socket is marked non-blocking; no connections are present to be accepted.

SEE ALSO

Library Functions - `bind()`, `connect()`, `listen()`, `socket()`

NAME

access – check accessibility of a file

SYNOPSIS

```
#include <file.h>

int access(path, mode)
char *path;
int mode;
```

DESCRIPTION

access checks the access permissions described by **mode** for the file named **path**. The real user and group ID of the current process is used in the check. It is useful in programs which run with the effective user ID set to something other than the current real user ID. Such programs can use access to check whether the real user has permission to access a resource.

mode is the bitwise-OR of any number of the following:

R_OK	check for read permission
W_OK	check for write permission
X_OK	check for execute permission
F_OK	check for existence of file

Note that access checks only the permission fields, not the file type.

DIAGNOSTICS

access returns -1 if the requested access permissions are not allowed or the file cannot be found, otherwise 0.

If the call fails **errno** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENOENT	The path name is too long, or the file does not exist.

`EPERM`	The path name contains a non-`ASCII` character.
`ELOOP`	Too many symbolic links were encountered when traversing the path.
`EROFS`	Write access is denied because file is on a read-only file system.
`ETXTBSY`	Write access is denied because the file is being executed.
`EACCES`	Requested access for file denied according to the mode of the file, or a directory in the path denied search permission.
`EFAULT`	**`path`** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - `chmod(), stat()`

NAME

`adjtime` – correct time to allow synchronization of the system clock

SYNOPSIS

```
#include <sys/time.h>

int adjtime(delta, olddelta)
struct timeval *delta;
struct timeval *olddelta;
```

DESCRIPTION

`adjtime()` adjusts the system's notion of the current time, as returned by `gettimeofday`, advancing or retarding it by the amount of time specified in the `struct timeval` (defined in `sys/time.h`) pointed to by **`delta`**.

The adjustment is effected by speeding up (if that amount of time is positive) or slowing down (if that amount of time is negative) the system's clock by some small percentage, generally a fraction of one percent. Thus, the time is always a monotonically increasing function. A time correction from an earlier call to `adjtime()` may not be finished when `adjtime()` is called again. If **`olddelta`** is not a `NULL` pointer, then the structure it points to will contain, upon return, the number of microseconds still to be corrected from the earlier call. If **`olddelta`** is a `NULL` pointer, the corresponding information will not be returned.

This call may be used in time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

Only the superuser may adjust the time of day.

RETURN

A 0 return value indicates that the call succeeded. A -1 return value indicates an error occurred, and in this case an error code is stored into the global variable **`errno`**.

ERRORS

EFAULT	**delta** or **olddelta** points outside the process's allocated address space. **olddelta** points to a region of the process' allocated address space that is not writable.
EPERM	The process's effective user ID is not that of the superuser.

SEE ALSO

System Calls- `gettimeofday()`

NAME

`arequest` – asynchronously do something to/from/with a file

SYNOPSIS

```
#include <asynchio.h>

arequest(request, fd, acbp, nargs, args )
int request, nargs, fd;
acb_t *acbp;
void **args;
```

DESCRIPTION

`arequest` performs asynchronous operations on the file **fd**. An asynchronous operation is performed by a separate thread of execution, in parallel with computation by the calling thread. When the asynchronous operation completes, the signal `acbp->a.evt.sig` is sent to the calling thread, with event data `acbp->a.evt.evtdata` if the signal is a reliable signal. If `acbp->a.evt.sig` is 0, no signal is sent.

The request is determined based on the value of **request**. The **nargs** argument indicates the number of arguments to the asynchronous operation; these arguments are passed in args. **nargs** does not include the file descriptor; it is only the number of arguments which are pointed to by **args**. **acbp** points to an asynchronous control buffer, used by the system to store the return value and **errno** result of the operation.

When the request is first submitted, `EINPROGRESS` is written to `acbp->a.errno_val`; this value may be polled to determine when the asynchronous operation completes. When the asynchronous operation completes, the return value from the operation is written into the area pointed to by `acbp->a.numbytes`; if an error occurred, the error number (**errno**) is also written into `acbp->a.errno_val`. The **errno** value is guaranteed to be written, so polling may occur, if one has the cycles to waste.

In addition, **acbp** can be used in a subsequent `acancel` operation to try and cancel the request, or in a subsequent `await` operation, to wait for pending asynchronous operations.

request has predefined values as follows:

ASYNCH_READ

```
int request, nargs, fd; acb_t *acbp;
struct {
  char *buf;
  off_t nbytes, offset;
  int whence;
} addl_args;
arequest(ASYNCHIO_READ, acbp, fd, 4, &addl_args)
```

An `lseek`, followed by a `read`, is performed to the file **fd**. The additional arguments to **arequest** should be a buffer address (**addl_args.buf**), a long count of bytes to be read (**addl_args.nbytes**), a long file position (**addl_args.offset**) and a whence value (**addl_args.whence**). The asynchronous operation is to perform an `lseek` with the parameters **fd**, **offset** and **whence**, and then perform the read with the arguments **fd**, **buff**, and **count**. The return is the return value of the corresponding `read`, unless the `lseek` fails, in which case the error from that operation is reported.

ASYNCH_WRITE

```
int request, nargs, fd; acb_t *acbp;
struct {
  char *buf;
  off_t nbytes, offset;
  int whence;
} addl_args;
arequest(ASYNCHIO_WRITE, acbp, fd, 4,& addl_args);
```

An `lseek`, followed by a `write`, is performed to the file **fd**. The additional arguments to `arequest` should be a buffer address (**addl_args.buf**), a long count of bytes to be written (**addl_args.nbytes**), a long file position (**addl_args.offset**) and a whence value (**addl_args.whence**). The asynchronous operation is to perform an `lseek` with the parameters **fd**, **offset** and **whence**, and then perform the `write` with the arguments **fd**, **buff**, and **count**. The return is the return value of the corresponding `write`, unless the `lseek` fails, in which case the error from that operation is reported.

CAUTION

`arequest` is a low-level asynch I/O facility used to implement `aread` and `awrite`. It should not be directly used by programs.

NOTE

When multiple asynchronous operations are being performed against a file, it is not possible to accurately determine the file pointer position. In general, all I/O performed on such a (block) file should be preceded by `lseek` with `L_SET`. Character special files are, of course, a different matter since seeks are constrained on them anyway.

`arequest` is present to support POSIX.4 (Draft 9) Asynchronous I/O. `arequest` may be removed in future versions of LynxOS. Those functions (`aread`, `awrite`, and `listio`) provide a much simpler interface to this functionality and are highly recommended.

DIAGNOSTICS

If `arequest` fails, -1 is returned.

If the call fails **`errno`** will contain one of the following:

`EINVAL`	**`request`** is not a known request.
`EINVAL`	`acbp->a.evt.sig` is an invalid signal number.
`EBADF`	**`fd`** does not refer to a valid file.
`ENOMEM`	No more memory is available for asynchronous requests or event storage.
`EAGAIN`	All the asynchronous I/O threads are currently busy.

SEE ALSO

System Calls - `acancel(), await(), read(), write(), lseek()`

NAME

`await` – wait for one of a set of asynchronous requests

SYNOPSIS

```
#include <asynchio.h>

int await(nacb, acbps)
int nacb;
acb_t **acbp;
```

DESCRIPTION

`await` suspends execution of the calling thread until one of the asynchronous operations described by the vector **acbp** completes. The size of the vector is **nacb** elements; NULL entries in the vector are ignored.

The **acb_t** structure contains the following fields, not necessarily in this order:

```
struct event evt;
int errno_val;
int retval;
```

The first asynchronous operation to complete will cause the thread to awaken; the index of that asynchronous control buffer address will be returned from the function.

NOTE

`await` is interruptible by signal delivery, and asynchronous requests often result in signal delivery. The clever programmer will ensure that a signal delivery will not cause await to fail due to the completion of one of the awaited operations.

DIAGNOSTICS

If `await` fails, - 1 is returned.

If the call fails **errno** will contain one of the following:

`EINTR`	A signal or event was delivered.
`EINVAL`	All of the elements in the vector **`acbp`** are NULL.
`EFAULT`	**`acbp`** is a bogus address.

SEE ALSO

System Calls - `arequest()`, `acancel()`

NAME

bdv_install,
bdv_uninstall – install and uninstall a block device

SYNOPSIS

```
int bdv_install(path, driver_ID, extra)
char *path;
int driver_ID, extra;

bdv_uninstall(bdevice_ID)
int bdevice_ID;
```

DESCRIPTION

bdv_install dynamically installs a block major device. It returns the major device ID. Once a major device is installed the mknod system call can be used to create any number of block special files for the minor devices that the driver supports. The device number for the mknod call can be determined by multiplying the major device number by 256 and adding the minor number.

path is the pathname to the information file for the major device. This file is loaded into the operating system data space and is made available to the device driver for the device. The driver is identified by the **driver_ID**. The driver ID is returned by the dr_install system call, if the driver has been dynamically installed. If the driver was pre-linked, the driver ID is its index in the driver configuration table. The drivers utility prints out all loaded or installed driver ID numbers.

extra is passed to the driver upon the device install. Its use is driver-specific.

bdv_uninstall will unload any major device. **bdevice_ID** is the ID number returned by bdv_install. A major device cannot be removed if any process has any one of its minor devices open.

These routines can only be called by processes whose effective user ID is 0 (super-user processes).

DIAGNOSTICS

`bdv_install` returns the major device ID if successful, or -1 if not.

If `bdv_install` fails **errno** will contain one of the following:

ENOMEM	There is not enough free memory in the system to load the information file.
ENOSPC	There are no more device slots.
ENODEV	No driver exists with driver number **driver_ID**, or the driver is for a different type of device.
ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
EPERM	The effective user ID of the calling process is not 0.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	Requested access for file denied according to the mode of the file, or a directory in the path denied search permission.
EFAULT	**Path** does not point to allocated memory in the process's address space.

`Bdv_uninstall` returns 0 if successful, or -1 if not.

If `bdv_uninstall` fails **errno** will contain one of the following:

EBADF	**bdevice_ID** is not a valid major block device ID.
EBUSY	Some process has this major device open.
EPERM	The effective user ID of the calling process is not 0.

SEE ALSO

System Calls - `dr_install(), mknod()`

NAME

`bind` – bind a name to a socket

SYNOPSIS

```
#include <types.h>
#include <socket.h>

bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

`bind` assigns a name to an unnamed socket. When a socket is created with `socket` it exists in a name space (address family) but has no name assigned. `bind` requests the **name**, be assigned to the socket.

NOTE

The rules used in name binding vary between communication domains.

DIAGNOSTICS

`bind` returns `0` if successful, or `-1` if not.

If the call fails, **errno** will contain one of the following:

EBADF	**s** is not a valid descriptor.
ENOTSOCK	**s** is not a socket.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EADDRINUSE	The specified address is already in use.
EINVAL	The socket is already bound to an address.
EACCES	The requested address is protected, and the current user has inadequate permission to access it.

`EFAUL`	The **`name`** parameter is not in a valid part of the user address space.

SEE ALSO

System Calls -	`connect()`, `listen()`, `socket()`, `getsockname()`

NAME

`brk, sbrk, lsbrk` – set or increment the break

SYNOPSIS

```
char *brk(addr)
char *addr;

char *sbrk(incr)
int incr;

char *lsbrk(incr)
long incr;

char *nsbrk(addr, incr)
char *addr;
long incr;
```

DESCRIPTION

The `brk` system call sets the end of the data space (the break) for a process to **`addr`**. **`addr`** is first rounded up to the next page boundary.

`sbrk` and `lsbrk` are used to move the break up by **`incr`**. `lsbrk` allows a long increment for large adjustments.

The `nsbrk` system call adjusts the break only if **`addr`** is equal to the current break. This comparison and adjustment is done atomically. Since **`incr`** can be either positive or negative, it is most useful in memory allocation code. This system call was added so that we could safely implement a version of `malloc` that returns memory to the system when it is freed.

DIAGNOSTICS

`brk` returns the new break if it was successfully set, or `-1` if not. `sbrk` and `lsbrk` return the old break (the beginning of the freshly allocated memory) if successful, or `-1` if not. `nsbrk` returns the old break or `-1` if **`addr`** was not equal to the current break.

If the call fails, `errno` will contain one of the following:

`ENOMEM`	The resource limit for the data segment size of the process was exceeded. This limit is set with `setrlimit`.
`ENOMEM`	The maximum data segment size for any process was exceeded.
`ENOMEM`	There is not enough memory in the system to grant the request.
`EFAULT`	The address passed to `nsbrk` is beyond the current break.
`EINVAL`	The address passed to `nsbrk` is less than the current break.

NOTE

`execve` sets the break when a program is loaded in and run, so `sbrk` is normally used to dynamically allocate more memory from the system.

SEE ALSO

System Calls - `execve()`, `getrlimit()`

Library Functions - `malloc()`, `free()`

NAME

`cdv_install`,
`cdv_uninstall` – install and uninstall a character device

SYNOPSIS

```
int cdv_install(path, driver_ID, extra)
char *path;
int driver_ID, extra;

cdv_uninstall(cdevice_ID)
int cdevice_ID;
```

DESCRIPTION

`cdv_install` dynamically installs a character major device. It returns the major device ID. Once a major device is installed the `mknod` system call can be used to create any number of character special files for the minor devices that the driver supports. The device number for the `mknod` call can be determined by multiplying the major device number by 256 and adding the minor number.

Path is the pathname to the information file for the major device. This file is loaded into the operating system data space and is made available to the device driver for the device. The driver is identified by the **`driver_ID`**. The driver ID is returned by the `dr_install` system call, if the driver has been dynamically installed. If the driver was pre- linked, the driver ID is its index in the driver configuration table. The `drivers` utility prints all loaded and installed driver ID numbers.

`extra` is passed to the driver upon the device install. Its use is driver specific.

`cdv_uninstall` will unload any major device. **`cdevice_id`** is the ID number returned by `cdv_install`. A major device cannot be removed if any process has any one of its minor devices open.

These routines can only be called by processes whose effective user ID is 0 (super-user processes).

DIAGNOSTICS

`cdv_install` returns the major device ID if successful, or -1 if not. If `cdv_install` fails **`errno`** will contain one of the following:

ENOMEM	There is not enough free memory in the system to load the information file.
ENOSPC	There are no more device slots.
ENODEV	No driver exists with driver number **`driver_ID`**, or the driver is for a different type of device.
ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
EPERM	The effective user ID of the calling process is not 0.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	Requested access for file denied according to the mode of the file, or a directory in the path denied search permission.
EFAULT	**`path`** does not point to allocated memory in the process' address space.

`cdv_uninstall` returns 0 if successful, or -1 if not. If `cdv_uninstall` fails **`errno`** will contain one of the following:

EBADF	**`cdevice_ID`** is not a valid major character device ID.
EBUSY	Some process has this major device open.

SEE ALSO

System Calls - `dr_install()`, `mknod()`

NAME

chcdev – set current control device

SYNOPSIS

```
chcdev(path)
char *path;
```

DESCRIPTION

chcdev changes the control device of the calling process to the given character special file. Thereafter, a call to open the unique control device interface file (usually called /dev/tty) will open the control device given. The control device of a process is inherited by any sub- processes created.

For interactive processes, the control device is usually set to a tty device, although any character special file may be used.

If **path** is zero, chcdev clears any current control device association. Processes which operate primarily as background servers or daemons usually clear the control device.

DIAGNOSTICS

chcdev returns 0 if the control device was successfully set, or -1 if not.

If the call fails **errno** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENOTCHR	The file is not a character special file.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	Requested access for file denied according to the mode of the file, or a directory in the path denied search permission.

`EFAULT`	**`path`** does not point to allocated memory in the process' address space.

NAME

`chdir` – set current working directory

SYNOPSIS

`chdir(path)`

`char *path;`

DESCRIPTION

`chdir` sets the current working directory to the directory named **`path`**. Whenever a pathname is examined by the system, the search for the file through the directory tree begins at either the root of the whole file system or at the current working directory. If the pathname begins with a slash character (/) searching begins at the root; otherwise searching begins at the current working directory.

The current working directory of a process is inherited by any sub-processes created.

DIAGNOSTICS

`chdir` returns 0 if the current working directory was successfully set, or -1 if not.

If the call fails **`errno`** will contain one of the following:

`ENOTDIR`	An element of the path is not a directory.
`ENAMETOOLONG`	The path name is too long.
`ENOENT`	The directory does not exist.
`ELOOP`	Too many symbolic links were encountered when traversing the path.
`EACCES`	Requested access for file denied according to the mode of the file, or a directory in the path denied search permission.
`EFAULT`	**`path`** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - `chroot()`

Library Functions - `getwd()`

NAME

chmod, fchmod – change the mode of a file

SYNOPSIS

```
#include <types.h>
#include <stat.h>

chmod(path, mode)
char *path;
int mode;

fchmod(fd, mode)
int fd;
int mode;
```

DESCRIPTION

chmod sets the access mode for the file named **path**. fchmod sets the access mode for the file open on file descriptor **fd**. The argument **mode** represents the new mode as the bitwise-OR of any number of the following constants:

S_ISVTX	keep text segment resident after execution
S_IREAD	read permission (owner)
S_IWRITE	write permission (owner)
S_IEXEC	execute permission (owner)

The text segment of an executable file is always sharable. If mode S_ISVTX is set the text segment of the executable file will remain in memory even if no process is using it. The bits S_IREAD, S_IWRITE, and S_IEXEC represent the three permission bits for the file owner. The permission values for the owning group and for other users can be obtained by shifting these three constants three or six bits to the right.

DIAGNOSTICS

chmod returns 0 if successful, or -1 if not.

If `chmod` fails, **`errno`** will contain one of the following:

`ENOTDIR`	An element of the path prefix is not a directory.
`ENAMETOOLONG`	The path name is too long.
`ENOENT`	The file does not exist.
`ELOOP`	Too many symbolic links were encountered when traversing the path.
`EACCES`	A directory in the path denied search permission.
`EFAULT`	**`path`** does not point to allocated memory in the process's address space.
`EROFS`	The file is on a read-only file system.

If `fchmod` fails, **`errno`** will contain one of the following:

`EBADF`	**`fd`** is not a valid file descriptor.
`EINVAL`	**`fd`** refers to a pipe.
`EROFS`	The file is on a read-only file system.

SEE ALSO

Utility Programs -	`chmod`
System Calls -	`open(), stat(), access()`

NAME

chown, fchown – change file ownership

SYNOPSIS

```
chown(path, uid, gid)
char *path;
int uid, gid;

fchown(fd, uid, gid)
int fd, uid, gid;
```

DESCRIPTION

chown changes the file ownership information recorded in the inode corresponding to the file named **path** to the given user ID **uid** and group ID **gid**. fchown performs the same operation as chown, but the file is specified by an open file descriptor instead of by name.

Changing the owner of a file is restricted to processes whose effective user ID is 0 (super-user processes). Changing the group of a file can also be done by super-user processes. A non-super-user process can change the group of a file as long as the process' effective user ID equals the user ID of the file, **uid** equals the file's user ID, and **gid** equals either the process's effective group ID or one of its supplementary groups IDs.

DIAGNOSTICS

chown and fchown return 0 if successful, -1 if not.

If an error occurs **errno** will contain one of the following values:

EPERM	The effective user ID of the calling process is not 0, and either the process's effective user ID is not equal to the file's user ID, **uid** is not equal to the file's user ID, or **gid** is not equal to the process's group ID or one of its supplementary groups IDs.
ENOTDIR	An element of the **path** prefix is not a directory.
ENAMETOOLONG	The **path** name is too long.

`ENOENT`	The file does not exist.
`ELOOP`	Too many symbolic links were encountered when traversing the **`path`**.
`EACCES`	A directory in the **`path`** denied search permission.
`EFAULT`	`path` does not point to allocated memory in the process' address space.
`EROFS`	The file is on a read-only file system.

If `fchown` fails, **`errno`** will contain one of the following:

`EBADF`	**`fd`** is not a valid file descriptor.
`EINVAL`	`fd` refers to a pipe.
`EROFS`	The file is on a read-only file system.

SEE ALSO

Utility Programs - `chmod, chown, ls`

System Calls - `chmod(), stat()`

NAME

`chroot` – change root directory

SYNOPSIS

```
chroot (path)
char *path;
```

DESCRIPTION

`chroot` sets the root directory of the calling process to the directory **path**. Thereafter, the operating system will begin searches at this directory for files whose pathnames begin with a slash character (/). All descendant processes from the current process inherit the new effective root. Note that once the root has been moved from the actual system root, the process cannot move it back.

`chroot` can only be called by processes whose effective user ID is 0 (superuser processes).

DIAGNOSTICS

`chroot` returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain one of the following:

EPERM	The effective user ID of the calling process is not 0.
ENOTDIR	An element of the **path** is not a directory.
ENAMETOOLONG	The **path** name is too long.
ENOENT	The directory does not exist.
ELOOP	Too many symbolic links were encountered when traversing the **path**.
EACCES	Requested access for file denied according to the mode of the file, or a directory in the **path** denied search permission.
EFAULT	**path** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - `chdir()`

NAME

`clock_settime`,
`clock_gettime`,
`clock_getres` – clock functions

SYNOPSIS

```
#include <time.h>

int clock_settime(clock_id, tp)
clockid_t clock_id;
const struct timespec *tp;

int clock_settime(clock_id, tp)
clockid_t clock_id;
struct timespec *tp;

int clock_getres(clock_id, res)
clockid_t clock_id;
struct timespec *res;
```

DESCRIPTION

The `clock_settime` function sets the clock **`clock_id`** to the value referenced by **`tp`**.

The `clock_gettime` function obtains the current value for the clock **`clock_id`** in the argument referenced by **`tp`**.

The `clock_getres` function obtains the resolution of the clock **`clock_id`** in the argument referenced by **`res`**, if **`res`** is not `NULL`. If **`res`** is `NULL`, it does not return the clock resolution.

The system-wide clock is visible to all processes, and represents the real time clock for the system. It has a **`clock_id`** of `CLOCK_REALTIME`. For the system-wide clock, the values returned by `clock_gettime()` and specified by `clock_settime()` represent the amount of time in seconds and nanoseconds, since 00:00 January 1, 1970.

NOTES

You must be superuser (UID 0) to call `clock_settime` with `CLOCK_REALTIME` as the **clock_id**.

DIAGNOSTICS

The functions `clock_settime`, `clock_gettime`, and `clock_getres` return `0` when successful. Otherwise a value of `-1` is returned indicating failure.

For following conditions, the `clock_settime`, `clock_gettime`, and `clock_getres` functions return `-1` and set **errno** to the corresponding value:

`EINVAL`	The **clock_id** argument does not correspond to any known clock.
`EPERM`	The effective user ID of the calling process is not 0.
`EFAULT`	**tp** or **res** is not a valid pointer.

CONFORMANCE

The clock functions `clock_settime`, `clock_gettime`, and `clock_getres` are implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Examples - `clock_settime.c, clock_gettime.c, clock_getres.c`

NAME

close – close a file

SYNOPSIS

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

DESCRIPTION

close removes a reference to a per-process file structure from the file descriptor table of the calling process. The reference is identified by the file descriptor **fd**, obtained from an open, socket, pipe, dup, or dup2 system call, or inherited from the parent process.

A file structure contains access modes set when a file is opened along with the present value of the seek pointer. A single file structure may be referenced by more than one descriptor via dup or dup2 or by the effect of a call to fork. When the last file descriptor that references a particular file structure is closed, the seek pointer value is lost and the file can only be accessed again by a new open.

All the open files of a process are closed upon exit. The system call fcntl can be used to arrange for selected files to be closed if the process calls execve.

Closing a file causes the system to cancel all pending asynchronous I/O for the file, as if acancel were called for each such request.

For POSIX.4 (Draft 9) message queue special files, the following exceptions apply:

- Messages sent by the process performing the close using the MSG_USE flag will be removed from the message file. If such messages cannot be immediately removed, the process will block until the messages are either delivered or removed. Messages sent with the MQ_MOVE or MQ_COPY flags will remain in the queue. Note that if the same process opens the file twice, the first close will cause all the MSG_USE messages to be removed, regardless of the existence of a second valid open file descriptor.

- After the last file descriptor that references a particular file structure is closed, one of the following actions will take place:
- If the file was created with the `MQ_PERSIST` option, the contents of the file will be unaffected
- If the file was created without the `MQ_PERSIST` option (the file is non-persistent by default), one should not rely on the persistence of the contents of the file. Note that in the case of a non-persistent message file, any existing messages are removed on the first `open` of the file.

Pending event notifications associated with the asynchronous message queue operations are unaffected.

For POSIX shared memory special files, the following exceptions apply:

- `close` behaves as usual for files, with additional effects on the last `close` of a shared memory special file. If the shared memory special file is persistent, the contents of the shared memory special file will persist. If this is the last `close` of a non-persistent shared memory special file, and the `close` will result in the file being unreferenced (not currently open by any process, and not mapped to any process), the contents of the file will be undefined after the `close` returns. If the file remains referenced after the last `close`, then the entire contents of the file will persist until the file becomes unreferenced.
- If this is the last `close` of any shared memory special file, and the `close` will result in the file being unreferenced, and the file's link count is zero, the file will be removed.

For a STREAMS file the following exceptions apply:

- A process that has registered to receive a `SIGPOLL` signal for events on a STREAMS file and later calls `close` with the file descriptor of the STREAMS file will be unregistered for events associated with the STREAMS file. If no other processes have STREAMS files opened associated with a particular `stream` then the `stream` will be dismantled. `close` waits up to 15 seconds for any output of the `stream` to drain before dismantling the stream when there are no signals posted for the `stream` and `O_NDELAY` is not set. `close` does not wait if there are signals pending or `O_NDELAY` is set.

DIAGNOSTICS

`close` returns 0 if successful, or -1 if not.

If the call fails, **`errno`** will contain:

`EBADF` **`fd`** is not a valid file descriptor.

`EINTR` A signal interrupted the `close` call.

SEE ALSO

System Calls - `dup(), execve(), fcntl(), flock(), open(), pipe(), unlink(), acancel()`

Library Calls - `aread(), awrite()`

Special calls that operate on POSIX message queue special files are:

System Calls - `mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()`

Library Functions - `mqgetpid()`

Special calls that operate on POSIX shared memory special files are:

System Calls - `mkshm(), shmmap(), shmunmap()`

NAME

`connect` – initiate a connection on a socket

SYNOPSIS

```
#include <types.h>
#include <socket.h>

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

DESCRIPTION

The parameter **s** is a socket. If it is of type SOCK_DGRAM, then this call permanently specifies the peer to which datagrams are to be sent; if it is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by **name** which is an address in the communications space of the socket. Each communications space interprets the **name** parameter in its own way.

DIAGNOSTICS

`connect` returns 0 if the connection or binding succeeds, or -1 if not.

If the call fails, **errno** will contain one of the following:

EBADF	**s** is not a valid descriptor.
ENOTSOCK	**s** is a descriptor for a file, not a socket.
EADDRNOTAVAIL	The specified address is not available on this machine.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EISCONN	The socket is already connected.
ETIMEDOUT	Connection establishment timed out without establishing a connection.
ECONNREFUSED	The attempt to connect was forcefully rejected.

`ENETUNREACH`	The network isn't reachable from this host.
`EADDRINUSE`	The address is already in use.
`EFAULT`	The **`name`** parameter specifies an area outside the process address space.
`EWOULDBLOCK`	The socket is non-blocking and the connection cannot be completed immediately.

SEE ALSO

System Calls - `accept(), socket(), getsockname()`

NAME

`dr_install, dr_uninstall` – install and uninstall a device driver

SYNOPSIS

```
#include <io.h>

int dr_install(path, type)
char *path;
int type;

dr_uninstall(id)
int id;
```

DESCRIPTION

`dr_install` provides a means for dynamically installing a device driver, either block or character. The type of the driver is identified by the **`type`** argument, and should be `BLOCKDRIVER` for a block driver or `CHARDRIVER` for a character driver. Loading a `BLOCKDRIVER` requires that a `CHARDRIVER` is loaded at the same time. The pathname of the driver is given in **`path`**. The name of the driver as displayed by the drivers utility is limited to the first 32 characters after the last '/' in **`path`**.

`dr_install` returns with the driver's ID number. In the case of a `BLOCKDRIVER`, `dr_install` returns a composite driver number with the `BLOCKDRIVER` driver number in the upper short and the `CHARDRIVER` driver number in the lower short.

`dr_uninstall` will unload any device driver. The argument **`id`** is the ID number returned by `dr_install`. A device driver cannot be uninstalled if any major devices are still installed that claim it as its driver (see `cdv_install(2)`).

Both of these routines can only be called by processes whose effective user ID is 0 (super-user processes).

DIAGNOSTICS

`dr_install` returns the driver ID number if successful, or -1 if not.
`dr_uninstall` returns 0 if successful, or -1 if not.

If `dr_install` fails **errno** will contain one of the following:

ENOMEM	There is not enough free memory in the system to load the driver.
ENOSPC	There are no more driver slots.
ENOTDIR	An element of the path is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
EPERM	The effective user ID of the calling process is not 0.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	Requested access for file denied according to the mode of the file, or a directory in the path denied search permission.
EFAULT	**path** does not point to allocated memory in the process's address space.

If `dr_uninstall` fails **errno** will contain one of the following:

EBADF	**id** is not a valid device driver ID.
EBUSY	There is a device using the driver.

SEE ALSO

Utilities -	`drivers`
System Calls -	`bdv_install()`, `cdv_install()`

NAME

dup, dup2 – duplicate a file descriptor

SYNOPSIS

```
int dup(oldfd)
int oldfd;

dup2(oldfd, newfd)
int oldfd;
int newfd;
```

DESCRIPTION

dup returns a file descriptor which references exactly the same file structure in the per-process file descriptor table as its argument **oldfd**. The original descriptor must be a valid open descriptor; it is left unchanged by the call. Because both file descriptors refer to the same file structure, file operations to either descriptor directly and immediately affect the condition of the other. The only characteristic of the original file descriptor not shared is the close-on-execute flag value. If an independent descriptor is desired, open should be used.

dup2 provides a duplicate descriptor in the manner of dup, except the number of the new descriptor is explicitly specified. If the requested descriptor is already in use, it is closed. dup2 is useful for duplicating descriptors into special descriptor numbers such as the standard input (0), output (1), or error (2) files. The range of descriptor numbers is system dependent, and can be obtained with the getdtablesize system call.

DIAGNOSTICS

If the file descriptor could not be duplicated, both dup and dup2 return -1.

If dup or dup2 fails **errno** will contain one of the following:

EBADF	**oldfd** or **newfd** is not a valid file descriptor.
EMFILE	The are no more free file descriptors.

NAME

`ekill` – send an event signal

SYNOPSIS

```
#include <events.h>

#include <proc.h>

int ekill(pid, evt_class, evt_data)
int pid;
int evt_class;
void *evt_data;
```

DESCRIPTION

`ekill` passes signal code of a POSIX 1003.4 Draft 9 event. `ekill` sends the event signal specified by **`evt_class`** along with the event data specified by **`evt_data`** to the process specified by **`pid`**. is passed on to the event handler. The event number must lie between `ECLOW` and `ECHI` (inclusive).

If **`pid`** is 0, the signal is sent to all other processes in the process group of the caller. If **`pid`** is -1, the signal is sent to all processes on the system except processes 0 and 1.

Otherwise, the **`pid`** is interpreted as a concatenation of a process ID and a thread ID built using `BUILDPID (process_id, thread_id)`. If the `thread_id` part of **`pid`** is zero, then **`pid`** refers to the entire process named by the `process_id` part of **`pid`**. If the `process_id` part of **`pid`** is zero, then the `thread_id` is interpreted within the current process.

A process can send an event to itself.

Event signals differ from standard signals in the following ways:

1. An event signal is queued for delivery and is never lost. Standard signals may be lost.
2. An event signal carries an additional data parameter which is passed to the event signal handler along with the signal number.

A signal handler for an event signal generated by the `ekill` function is set with the `sigaction` function. The signal action varies depending on whether the **SA_SIGINFO** bit is set for the signal in the `sa_flags` member of the `sigaction` structure used in `sigaction`.

If the **SA_SIGINFO** flag is not set for the signal, the signal handler function is set with the `sa_handler` member of the `sigaction` structure. This function is entered as:

```
void function(int signo);
```

If the **SA_SIGINFO** flag is set for the signal, the signal handler function is set with the `sa_sigaction` member of the `sigaction` structure. The function is entered as:

```
void function(int signo, siginfo_t *info, void *context);
```

signo is the signal number for the signal being delivered. info is a pointer to a **siginfo_t** structure defined in `<signal.h>`. The context argument is not used by LynxOS. See `sigaction(2)` for more information on the `siginfo_t` structure.

SA_D9EV flag must be set in the `sa_flags` field of the `sigaction` structure when `sigaction` is called.

DIAGNOSTICS

`ekill` returns 0 if successful, or -1 if not.

If the call fails `errno` will contain one of the following:

ESRCH	process **pid** does not exist.
EINVAL	**evt_class** is invalid.

SEE ALSO

System Calls - `getpid()`, `getpgrp()`, `killpg()`, `sigvec()`, `sigaction()`

NAME

`evtpoll`, `esigpoll` – wait for one of a set of signals

SYNOPSIS

```
#include <signal.h>
int evtpoll (mask, timeout, value, signumber)
sigset_t *mask;
struct timespec *timeout;
void **value;
int *signumber;

int esigpoll (mask, timeout, value, signumber)
sigset_t *mask;
struct timespec *timeout;
void **value;
int *signumber;
```

DESCRIPTION

`evtpoll` waits for one of the signals in **`mask`** to be delivered to the thread. When such a signal is delivered, its number will be stored at the location pointed to by **`signumber`** if **`signumber`** is not NULL. If the signal is an event signal and has an associated data value, the data value will be stored at the location pointed by **`value`** if **`value`** is not NULL. If **`timeout`** is not NULL, it indicates a maximum time interval to wait for a signal to occur. If the time elapses before a signal arrives, then `evtpoll` will return `-1` with **`errno`** set to `EAGAIN`. If **`timeout`** is NULL, the call will never time out. `The mask` can contain both event signals and other signals.

When an awaited signal arrives at the thread, the thread is awakened and zero is returned. Unlike `sigwait`, `evtpoll` will never return due to a signal that was directed at the process, unless the calling thread is the initial thread, and no other thread is performing a `sigwait` for the signal.

`evtpoll` has priority over signal handlers and signal masks, but not over signal ignoring. Thus, if a thread is performing an `evtpoll` for a signal, it will be awakened when that signal arrives even if the thread has the signal blocked or has a handler set up for it, but not if the thread is ignoring the signal.

The proper usage of `evtpoll` is to first call `sigprocmask` to block the set of signals on which `evtpoll` will be waiting, and then call `evtpoll` on that same set.

`esigpoll` performs exactly the same function as `evtpoll`.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

Obviously, `SIGTHREADKILL` should not be waited for using `evtpoll` if the process consists of multiple, cancelable threads.

One of the most common programming errors is the use of the mask itself, rather than the address of the mask, in calls to `evtpoll`. The following code has a bug and will not work:

```
int sigGot;
long mask;
mask = 1 << (SIGUDEF28 - 1);
if (evtpoll(mask, NULL, NULL, &sigGot) < 0)
  perror("evtpoll");
printf("Got signal %d\n", sigGot);
```

The proper code is the following:

```
int sigGot;
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGUDEF28);
if (evtpoll(&mask, NULL, NULL, &sigGot) < 0)
  perror("evtpoll");
printf("Got signal %d\n", sigGot);
```

DIAGNOSTICS

`evtpoll` returns `0` if a signal arrived, or `-1` if not.

If the call fails, **`errno`** will contain one of the following:

EFAULT	**mask**, **timeout**, **value**, or **signumber** does not point to allocated memory in the process' address space.
EAGAIN	**timeout** is not NULL, and no signal specified by mask was delivered within the specified timeout period.

SEE ALSO

System Calls -	`kill()`, `sigvec()`, `ekill()`, `sigsetmask()`, `sigblock()`
Library Functions -	`evtraise()`, `sigemptyset()`, `sigaddset()`

NAME

`execve` – execute a file

SYNOPSIS

```
execve(path, argv, envp)
char *path, *argv[], *envp[];
```

DESCRIPTION

`execve` loads the program named **`path`** over the calling program and runs it. Since the new program is the same process as the calling program, there is no return from a successful call. The new program must be either an executable object file or a source file for some interpreter.

An executable object file contains a header, a text segment (machine code), and a data segment (initialized data). The header contains the size of the bss segment (uninitialized variable space). The text segment and data segment are loaded into the process' text and data segment space; space for the bss segment is allocated and cleared before execution.

An interpreter source file begins with a line of the form

```
#! <name of interpreter> <single optional argument>
```

Note: The `#` character must be the first character on the first line; the `!` character must immediately follow.

When such a file is given to `execve`, the interpreter named is loaded, and the name of the source file is passed as an argument. If the single optional argument is given in the source file header, it is placed before the source file name. Additional parameters given in the original call will be placed after the source file name.

Argument **`argv`** is an array of pointers to C-style character strings; the last element in the array must be a null pointer. The first element in the array is normally the name of the program being executed. Argument **`envp`** is a similarly structured array which contains the environment for the new program.

The newly-loaded program is passed the argument and environment lists as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

`argc` contains the number of elements in `argv` not including the last element (the null). The contents of the strings referenced in the `execve` call are passed unmolested to the new program.

All open file descriptors remain open across the `execve`, unless the close-on-execute flag has been set with the `fcntl` system call. All caught signals are reset to their default, but ignored signals remain ignored, and masked signals remain masked.

DIAGNOSTICS

If `execve` works it will not return, otherwise it returns -1.

If `execve` fails, **`errno`** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	A directory in the path denied search permission.
EACCES	The file is not a regular file.
EACCES	The file mode denies execute permission.
ENOEXEC	The file header contains the wrong magic number.
ETXTBSY	The file is already open for reading or writing.
ENOMEM	The program requires more memory than is available in the system.
EFAULT	**`path`**, **`argv`**, or **`envp`** do not point to allocated memory in the process' address space.

`EFAULT` The file header is wrong.

SEE ALSO

System Calls - `_exit()`, `fork()`, `fcntl()`

Library Functions - `execl()`, `getenv()`

NAME

`_exit` – terminate the calling process

SYNOPSIS

```
_exit(status)

int status;
```

DESCRIPTION

`_exit` causes the calling process to terminate. All open file descriptors of the process are closed. **`status`** is returned to the parent of the calling process through the `wait` system call.

If the calling process has any child processes, they are inherited by process 1 (**`init`**).

NOTE

If a program has done any stream I/O then the library routine `exit` should be called to terminate. `exit` flushes the stream buffers and calls `_exit`. A return from `main` in a C program effectively calls `exit`.

SEE ALSO

Library Functions - `exit(), fclose()`

NAME

`fcntl` – file control

SYNOPSIS

```
#include <types.h>
#include <fcntl.h>

fcntl(fd, cmd, arg)
int fd, cmd, arg;
```

DESCRIPTION

`fcntl` provides various controls over either a file descriptor or the underlying file structure. The argument **fd** is the active file descriptor to be manipulated. The operation is given in **cmd** as follows:

F_DUPFD — Return a duplicate file descriptor with a value greater than or equal to **arg** (see `dup`).

F_GETFD — Return the close-on-execute flag for file descriptor **fd**. If the least significant bit is "on" the file will be closed on a successful `execve`.

F_SETFD — Set the close-on-execute flag for file descriptor **fd** to the least significant bit of **arg**.

F_GETFL — Get the file descriptor mode flags.

F_SETFL — Set the file descriptor mode flags.

F_GETLK — Get the first lock which would block the lock specified in the `flock` structure pointed to by **arg**. The information retrieved overwrites the information in the flock structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except for the lock type which will be set to `F_UNLCK`.

F_SETLK — Set or clear an advisory record lock according to the `flock` structure pointed to by **arg**. `F_SETLK` is used to establish shared (`F_RDLCK`) and exclusive (`F_WRLCK`)

	locks, or to remove either type of lock (`F_UNLCK`). If a shared or exclusive lock cannot be created, `fcntl` will return with an error value of -1.
`F_SETLKW`	This **`cmd`** is the same as `F_SETLK` except that if a shared or exclusive lock is blocked by other locks, the process will be suspended until the lock can be created.
`F_GETOWN`	Get the process ID or process group currently receiving signals from the driver. A process group is indicated by a negative return value.
`F_SETOWN`	Set the process ID or process group that is to receive signals from the driver. A process ID is specified by a positive **`arg`** value, and a process group is specified by a negative **`arg`** value.

The file descriptor mode flags that can be set by `F_SETFL` and retrieved by `F_GETFL` are the following:

`FNDELAY`	Don't wait on a `read` or `write` call. If no data or buffer space is available, return -1 with **`errno`** set to `EWOULDBLOCK`.
`FAPPEND`	Cause all writes to append to the end of the file.

Advisory record locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access. The files remain completely accessible to processes that do not attempt locks.

There are two types of locks: shared (`F_RDLCK`) and exclusive (`F_WRLCK`). More than one process may hold a shared lock for a particular segment of a file at any given time, but multiple exclusive locks, or both shared and exclusive locks, may not exist simultaneously on any segment.

A process must have opened the file with read access in order to create a shared lock; with write access in order to create an exclusive lock.

A shared lock can be upgraded to an exclusive lock, and vice versa, simply by using `F_SETLK` or `F_SETLKW` with the appropriate lock type. The previous lock will be removed and the new lock created, possibly after other processes have gained and released the lock.

If **cmd** is F_SETLKW and the lock cannot be created immediately, then the process will block until the lock can be created. Processes waiting for locks are interrupted by signals. Setitimer may thus be used to provide a timeout facility.

A record lock is described by the flock structure, defined in fcntl.h as follows:

```
struct flock {
  short l_type;   /* FRDLCK, FWRLCK, or FUNLCK */
  short l_whence;   /* Flag for starting offset */
  long l_start;   /* Relative offset in bytes */
  long l_len;   /* Size; if 0 then EOF */
  short l_sysid;
  short l_pid;   /* Process ID of the process
    holding the lock */
};
```

The flock structure describes the type (l_type), starting offset (l_whence), relative offset (l_start), and size (l_len) of the segment of the file to be locked or unlocked. The relative offset is measured from the start of the file, the current position, or the end of the file depending on whether l_whence is SEEK_SET, SEEK_CUR, or SEEK_END. The process ID field (l_pid) is only used with F_GETLK to return the description of a lock held by another process.

Locks may start and extend beyond the current end-of-file, but may not be negative relative to the beginning of the file. A lock may be set to always extend to the end-of-file by setting l_len to zero. If such a lock also has l_whence and l_start set to zero, the entire file will be locked. Changing or unlocking a segment from the middle of a larger locked segment leaves two smaller segments at either end. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect. All locks on a given file held by a given process are removed when the file is closed, or when the process terminates. Locks are not inherited by child processes.

A potential for deadlock occurs if a process holding a lock is suspended by attempting to lock a segment that is locked by another process. Therefore, the F_SETLKW command scans for a deadlock prior to waiting for a lock. An error is returned if waiting for a lock would cause deadlock.

For POSIX message queue special files, the following exceptions apply:

`F_SETFL`	Only the `O_NONBLOCK` value of the file status flags is supported, but attempts to set other flags will not result in any errors returned by the system. Note that `O_NONBLOCK` is the same as `FNDELAY`.
`F_GETLK`	Same functionality as listed above.
`F_SETLK`	Same functionality as listed above.
`F_SETLKW`	Same functionality as listed above.

NOTE

Interaction between locks and buffered input/output systems may produce anomalous situations. Careful (even aggressive) use of buffer flushing requests is recommended.

DIAGNOSTICS

Upon successful completion, the value returned by `fcntl` depends on **`cmd`** as follows:

`F_DUPFD`	A new file descriptor.
`F_GETFD`	Value of flag (only the low-order bit is defined).
`F_SETFD`	0.
`F_GETFL`	Value of file flags.
`F_SETFL`	0.
`F_GETLK`	0.
`F_SETLK`	0.
`F_SETLKW`	0.
`F_GETOWN`	Value of descriptor owner.
`F_SETOWN`	0.

Otherwise, `fcntl` returns -1.

If `fcntl` fails, **`errno`** will contain one of the following:

EBADF	**fd** is not a valid open file descriptor.
EBADF	**cmd** is F_SETLK or F_SETLKW and the process does not have the appropriate read or write permissions on the file.
EMFILE	There are no more free file descriptors.
EINVAL	**cmd** is F_DUPFD and **arg** is not a valid file descriptor number.
EINVAL	**cmd** is F_GETLK, F_SETLK, or F_SETLKW and the `flock` structure pointed to by **arg** is not valid.
EINVAL	**cmd** is not one of the valid commands.
EFAULT	**cmd** is F_GETLK, F_SETLK, or F_SETLKW and **arg** does not point to allocated memory in the process address space.
EACCES	**cmd** is F_SETLK, the lock type (`l_type`) is F_RDLCK (shared lock), and the segment of the file to be locked already has an exclusive lock held by another process, or the lock type is F_WRLCK (exclusive lock) and the segment of the file to be locked already has either a shared or exclusive lock held by another process.
ENOLCK	**cmd** is F_SETLK or F_SETLKW, a lock is being created (F_RDLCK or F_WRLCK), and there are no more file lock entries available.
EDEADLK	**cmd** is F_SETLKW, the lock is blocked by another process' lock, and suspending the calling process would cause a deadlock situation to occur.
EDEADLK	**cmd** is F_SETLK or F_SETLKW, a lock is being removed (F_UNLCK), and there are no more file lock entries available.

SEE ALSO

System Calls - `close()`, `execve()`, `getdtablesize()`, `lockf()`, `open()`

Special calls that operate on POSIX message queue special files are:

System Calls - `mkmq()`, `mqsend()`, `mqreceive()`, `mqsetattr()`, `mqgetattr()`, `mqpurge()`, `mqgetpid()`, `msgalloc()`, `msgfree()`, `mqputevt()`, `mqgetevt()`

NAME

fdatasync – synchronize a file's data

SYNOPSIS

```
#include <unistd.h>

int fdatasync(fildes)
int fildes;
```

DESCRIPTION

fdatasync is used to synchronize all data associated with the open file descriptor **fildes** to the storage device associated with this file descriptor. All currently queued I/O operations associated with the file descriptor **fildes** are forced to the synchronized I/O data integrity completion state. A synchronized I/O data integrity completion state implies that all read and write operations have successfully completed, or are diagnosed as unsuccessful.

DIAGNOSTICS

fdatasync returns 0 upon successful completion. On failure it returns -1 and sets **errno** as:

EBADF	The **fildes** argument is not a valid file descriptor.
EINVAL	Synchronized I/O for this file is not supported.

If any of the queued I/O operations fail, fdatasync returns the error conditions defined for read and write.

CONFORMANCE

fdatasync is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - fsync(), read(), write()

NAME

`flock` – advisory file locks

SYNOPSIS

```
#include <file.h>

flock(fd, command)
int fd;
int command;
```

DESCRIPTION

`flock` locks or unlocks a file described by file descriptor **`fd`** with an advisory lock. The command should be one of the following:

`LOCK_SH`	lock with a shared lock.
`LOCK_EX`	lock with an exclusive lock.
`LOCK_UN`	unlock.

If any of the above values are OR-ed with `LOCK_NB`, the call to lock will not block if the lock is currently unattainable.

There can be only one exclusive lock on a file at any one time. An exclusive lock precludes any shared locks as well. If no exclusive lock is in effect, any number of shared locks may be set.

Since these are advisory locks, a lock on a file will guarantee consistency only if all processes using the file cooperate and request locks.

NOTE

These locks operate on files and not on file descriptors. Calling `flock` on a file descriptor that has been duplicated through `dup`, `dup2`, or `fork` will not conflict with a lock created using the original file descriptor, but instead will set up multiple references to a single lock. If a process holding a lock on a file forks and the child (parent) explicitly unlocks the file, the parent (child) will lose its lock.

DIAGNOSTICS

`flock` returns `0` if the lock was granted, otherwise `-1`.

If the call fails **`errno`** will contain one of the following:

`EBADF`	**`fd`** is not a valid file descriptor.
`EINVAL`	**`fd`** refers to a pipe.
`EAGAIN`	The `LOCK_NB` flag was set, and the file is locked.

NAME

`fork` – create a new process

SYNOPSIS

`int fork()`

DESCRIPTION

`fork` creates a child process that is an exact copy of the calling process except for the few differences noted below. `fork` returns the process ID number of the child process to the calling process, and returns 0 to the child process. Execution resumes in both processes at the point immediately following the call to `fork`:

```
if (fork()) {
  /* this is the parent */
  ..
}
else {
  /* this is the child */
  ...
}
```

The child process (like every process on the system) has a unique process ID. It can use the system calls `getpid` and `getppid` to determine the process ID numbers of itself and its parent, if necessary.

All shared memory of the parent is likewise shared by the child. All signal handling arrangements remain the same in the child as in the parent.

The child process has its own file descriptor table, but the descriptors describe the same file structures as the parent's file descriptors. The `seek` pointer and mode flags for the files are therefore shared with the parent process.

DIAGNOSTICS

`fork` returns -1 if a new process could not be created.

If the call fails, **`errno`** will contain one of the following:

`EAGAIN`	The maximum number of process that can be running on the system (`NPROC`) has already been reached.
`ENOMEM`	There is not enough free memory in the system to copy the parent's unsharable memory.

SEE ALSO

System Calls - `execve()`

NAME

`fsync` – synchronize a file's state

SYNOPSIS

```
#include <unistd.h>

int fsync(fildes)
int fildes;
```

DESCRIPTION

`fsync` is used to synchronize all information about the open file descriptor **`fildes`** to the storage device associated with this file descriptor. All currently queued I/O operations associated with the file descriptor fildes are forced to the synchronized I/O file integrity completion state. A synchronized I/O file integrity completion state implies the following:

- all `read` and `write` operations have successfully completed, or are diagnosed as unsuccessful
- all file attributes relative to the I/O operation (including access time, modification time, status change time) are successfully transferred.

DIAGNOSTICS

`fsync` returns `0` upon successful completion. On failure it returns `-1` and sets **`errno`** as:

`EBADF`	The **`fildes`** argument is not a valid file descriptor.
`EINVAL`	Synchronized I/O for this file is not supported.

If any of the queued I/O operations fail, `fsync` returns the error conditions defined for `read` and `write`.

CONFORMANCE

`fsync` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `fdatasync(), read(), write()`

NAME

`fsync` – sync a disk file

SYNOPSIS

```
fsync(fd)
int fd;
```

DESCRIPTION

`fsync` ensures that any memory-resident data of the file open with descriptor **`fd`** is written to the device on which the file resides.

DIAGNOSTICS

`fsync` returns 0 if successful, or -1 if not.

If the call fails, **`errno`** will contain one of the following:

`EBADF`	**`fd`** is not a valid file descriptor.
`EINVAL`	**`fd`** is not a regular file.

SEE ALSO

Utility Programs - `sync`

System Calls - `sync()`

NAME

`ftruncate` – truncate a file to a specified length

SYNOPSIS

```
#include <unistd.h>

int ftruncate(fildes, length)
int fildes;
off_t length;
```

DESCRIPTION

`ftruncate` truncates a file to a specified length. The **`fildes`** argument is the file descriptor for a regular file or a shared memory object, which must be open for writing. The **`length`** argument specifies the desired size of the file. If the current size of the file exceeds **`length`**, the extra data in the file is discarded. If the size of the file is smaller than **`length`**, then the file is extended. The extended area is filled with zero bytes.

A POSIX.1b shared memory object created with the `shm_open` function always has a zero size. Hence `ftruncate` is the only way to set the size of a POSIX.1b shared memory object.

DIAGNOSTICS

`ftruncate` sets the size of the specified file when successful, and returns `0`. On failure it returns `-1` and sets **`errno`** as:

`EBADF`	The **`fildes`** argument is not a valid file descriptor or shared memory object.
`EINVAL`	The **`fildes`** argument refers to a file which may not be truncated.
`EROFS`	The specified file resides on a read-only file system.

CONFORMANCE

`ftruncate` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `open(), shm_open()`

NAME

`getdents` – read directory entries into a file system independent format

SYNOPSIS

```
#include <dirent.h>

int getdents(fd, buff, count)
int fd;
char *buff;
unsigned count;
```

DESCRIPTION

`getdents` attempts to read count bytes from the directory associated with the open file descriptor **`fd`** and to format them as file system independent directory entries in the buffer pointed to by **`buff`**. A file system independent directory entry resides in a dirent structure defined in `<dirent.h>` as follows:

```
struct dirent {
    long d_ino;
    off_t d_off;
    unsigned short d_reclen;
    char d_name[1];
};
```

The `d_ino` field is the `inode` number of the entry and is unique for every entry in the file system. The `d_off` field is the offset of the entry in the actual file system directory. The `d_name` field is the first character in the name of the entry. This name is null terminated and may have, at most, `MAXNAMLEN` characters. This means that file system independent directory entries are of variable length. The `d_reclen` field is the length of the entry in bytes. The length is always a multiple of four so that the next entry always begins on a long boundary.

Since the file system independent directory entries are of variable length, the actual number of bytes returned will often be less than **`count`**.

On devices capable of seeking, `getdents` begins reading at the `seek` pointer associated with **`fd`**. Upon return from `getdents`, the `seek` pointer is incremented to point to the next entry.

DIAGNOSTICS

`getdents` returns the actual number of bytes read if successful, or -1 if not. A return value of 0 indicates that the end of the directory has been reached.

If `getdents` fails **`errno`** will contain one of the following:

`EBADF`	**`fd`** is not a valid file descriptor.
`EFAULT`	**`buff`** does not point to allocated memory in the process' address space.
`EINVAL`	**`count`** is not large enough for one directory entry.
`ENOENT`	The `seek` pointer associated with **`fd`** is not located at a valid entry.
`ENOTDIR`	**`fd`** does not refer to a directory.
`EIO`	An I/O error occurred while accessing the file system.

NAME

`getdtablesize` – get the file descriptor table size

SYNOPSIS

`int getdtablesize()`

DESCRIPTION

`getdtablesize` returns the size of the per-process file descriptor table. A file descriptor is an index into this table. No process may have more files open than the size of the table.

NAME

`getgid`, `getegid` – get real or effective group ID

SYNOPSIS

```
int getgid()
int getegid()
```

DESCRIPTION

The routines `getgid` and `getegid` return the current real or effective group ID number of the calling process.

When a program with the `S_ISGID` (set group ID) bit is executed, the effective group is changed to the owning group of program. The program can discover the real group ID with `getgid`.

SEE ALSO

System Calls - `setregid()`

NAME

getgroups, setgroups – get and set group membership list

SYNOPSIS

```
getgroups(gcount, glist)
int *gcount;
int *glist;

setgroups(gcount, glist)
int gcount;
int *glist;
```

DESCRIPTION

In addition to the group ID associated with each user ID in the password file (/etc/passwd), a set of groups can be assigned to each user ID through the group file, /etc/group. Each system group is given a name and a list of member user IDs in this file. The setgroups system call is used by login to establish the group list according to /etc/group for the login shell. The group list is inherited by all child processes.

The group IDs are put into an array at address **glist**. The number of groups is passed as **gcount** and may be as large as NGROUPS (defined in proch). Only privileged processes (those with effective user ID of zero) can issue a call to setgroups.

getgroups returns the current group list in **glist**. The number of available entries in the array is passed in the integer pointed to by **gcount**; **gcount** is set by the routine to the actual number of groups. No more than NGROUPS groups will ever be returned.

DIAGNOSTICS

If getgroups succeeds, it returns 0 and places the number of groups in **gcount** and the actual list in **glist**. Otherwise, getgroups returns -1 and places one of the following values in **errno**:

EFAULT	Either **gcount** or **glist** points to memory outside of the process address space.

`EFAULT`	**`glist`** is not large enough to hold the number of groups requested in **`gcount`**.

If `setgroups` succeeds, it returns 0. Otherwise it returns -1 and **`errno`** is set to one of the following values:

`EFAULT`	**`glist`** points to memory outside the process address space.
`EPERM`	The calling process has a non-zero user ID (is not privileged).
`EINVAL`	**`gcount`** is larger than `NGROUPS`.

SEE ALSO

Utility Programs -	`groups`
System Calls -	`setregid(), getregid()`

NAME

gethostname, sethostname – get and set network host name

SYNOPSIS

```
gethostname(name, len)
char *name;
int len;

sethostname(name, len)
char *name;
int len;
```

DESCRIPTION

gethostname copies the current host name to the string **name**, with a maximum string length of **len**. The string is terminated by a null.

sethostname sets the host name to **name**. The length of this string is given by **len**. This system call is only available to processes running with effective user ID 0 (super-user processes).

DIAGNOSTICS

sethostname and gethostname return 0 if the host name was set, or -1 if not.

If a call fails **errno** will contain one of the following:

EFAULT	**name** does not point to allocated memory in the process' address space.
EINVAL	**len** was not big enough to hold the host name for gethostname.
EPERM	The effective user ID of the calling process is not 0.

NAME

getitimer, setitimer – get or set an interval timer

SYNOPSIS

```
#include <time.h>

getitimer(which, val)
int which;
struct itimerval *val;

setitimer(which, newval, oldval)
int which;
struct itimerval *newval, *oldval;
```

DESCRIPTION

Each process has three interval timers, for real time, virtual time, and profiling. `setitimer` sets a selected timer to the value pointed to by **newval**, and writes the old value at **oldval** if **oldval** is not zero. `getitimer` writes the current value of one of the timers at **val**.

`which` is used to specify a timer and can be one of the following:

ITIMER_REAL	for the real time timer.
ITIMER_VIRTUAL	for the virtual time timer.
ITIMER_PROF	for the profile timer.

The value of a timer is defined (in `time.h`) by the **itimerval** structure:

```
struct itimerval {
  struct timeval it_interval;
  struct timeval it_value;
};
```

`it_value` is the time before the timer sends its next alarm. If it is zero, the timer is disabled. When the timer expires the count is reloaded with `it_interval`. If `it_interval` is zero, the timer will be disabled after it expires.

A timer value which is non-zero, but less than the amount of time of one system clock tick, is rounded up to a single tick.

The real-time timer decrements all the time. It sends a `SIGALRM` when it counts down to zero.

The virtual timer decrements only when the process is running user code. It sends a `SIGVTALRM` when it counts down to zero.

The profile timer decrements only when the process is running user or system code. It sends a `SIGPROF` when it counts down to zero.

DIAGNOSTICS

`setitimer` and `getitimer` return 0 if the timer was set, or -1 if not.

If a call fails **errno** will contain one of the following:

EFAULT	**val**, **oldval**, or **newval** do not point to allocated memory in the process' address space.
EINVAL	One of the times specified was too big.
EINVAL	`which` is not one of `ITIMER_REAL`, `ITIMER_VIRTUAL`, or `ITIMER_PROF`.

SEE ALSO

System Calls - `sigvec(), gettimeofday()`

NAME

getmsg – get the next message off a stream

SYNOPSIS

```
#include <stropts.h>

int getmsg(fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int *flags;
```

DESCRIPTION

getmsg retrieves the contents of a message at the stream head read queue from a STREAMS file. getmsg then places the contents (either a data part, a control part, or both) into buffers specified by the user. The data and/or control parts of the message are placed into separate buffers. The STREAMS module that generates the message defines the semantics of each part.

The **fd** argument is a file descriptor referencing an open stream. **ctlptr** and **dataptr** point to **strbuf** structures which contain:

```
int maxlen;      /* maximum buffer length */
int len;         /* length of data        */
char *buf;       /* ptr to buffer          */
```

buf points to a buffer into which the data or control information is placed. maxlen indicates the maximum number of bytes buf can hold. On return, len comprises the number of bytes of data or control information received; a 0 indicates a zero-length control or data part and a -1 indicates no data or control information is present. **flags** specifies the message type and may be set to 0 or RS_HIPRI.

cltptr holds the control part from the message and **dataptr** holds the data part from the message. The control/data part of the message is not processed if **ctlptr** (or **dataptr**) is NULL or maxlen is -1; it is left on the stream head read queue. In this case len is set to -1.

If maxlen is set at 0, and there is a zero-length control/data part, the zero-length portion is deleted from the read queue and len is set to 0. If maxlen (in **ctlptr** or **dataptr**) is less than the control or data part of the message, maxlen bytes are retrieved. The balance of the message remains on the stream head read queue and a non-zero value is returned.

When information is retrieved from a priority message, **flags** is set to RS_HIPRI on return.

The getmsg default is to process the first message available on the stream head read queue. In order to override the default and have getmsg process only priority messages, the user must set **flags** to RS_HIPRI.

If O_NDELAY is not set, getmsg blocks until a message is available on the stream head read queue. If O_NDELAY is set and a message is not on the read queue, getmsg fails and sets **errno** to EAGAIN.

getmsg operates in this manner, even after a hangup occurs on the stream from which messages are retrieved, until the stream head read queue is empty. Then, getmsg will set len to 0 in **ctlptr** and **dataptr**.

DIAGNOSTICS

The following values are returned:

a non-negative value
: Successful completion.

0
: A full message was read successfully.

MORECTL
: More control information is queued to be retrieved.

MOREDATA
: More data is queued to be retrieved.

MORECTL|MOREDATA
: Both types of information remain. Subsequent calls by getmsg will retrieve the remainder.

-1
: System call failure; **errno** is set.

Possible values for **errno** are:

EAGAIN	The O_NDELAY flag is set, and there are no messages.
EBADF	**fd** is not a valid file decriptor.
EBADMSG	The queued message is not valid for getmsg.
EFAULT	**ctlptr**, **dataptr**, or **flags** points outside the allocated address space.
EINTR	A signal was caught during the getmsg system call.
EINVAL	An illegal value was specified in **flags**, or the **fd**-referenced stream is linked under a multiplexer.
ENOSTR	**fd** is not associated with a stream.

In addition, getmsg will fail if a STREAMS error message is received at the stream head before the call to getmsg takes place. The returned error represents the value contained in the STREAMS error message.

SEE ALSO

System Calls - poll(), putmsg(), read(), write()

NAME

`getpagesize` – get the system's memory page size

SYNOPSIS

`int getpagesize()`

DESCRIPTION

`getpagesize` returns the number of bytes in a memory page.

SEE ALSO

System Calls - `brk()`

NAME

`getpeername` – get name of connected peer

SYNOPSIS

```
getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

`getpeername` returns the name of the peer connected to socket **s**. The **namelen** parameter should be initialized to indicate the amount of space pointed to by **name**. On return it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails. The call succeeds unless:

EBADF	The argument **s** is not a valid descriptor.
ENOTSOCK	The argument **s** is a file, not a socket.
ENOTCONN	The socket is not connected.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
EFAULT	The **name** parameter points to memory not in a valid part of the process address space.

SEE ALSO

System Calls - `bind(), socket(), getsockname()`

NAME

getpgrp, setpgrp – get or set the process group

SYNOPSIS

```
int getpgrp(pid)
int pid;

setpgrp(pid, pgrp)
int pid;
int pgrp;
```

DESCRIPTION

getpgrp returns the process group ID for the process with ID **pid**. If **pid** is 0, the process group ID of the calling process is returned.

setpgrp sets the process group ID of process **pid** to **pgrp**. If **pid** is zero, the process group ID of the calling process is set. If **pgrp** is zero and **pid** refers to the calling process, setpgrp behaves identically to setsid, except that the return value is always zero.

Every process is part of a process group. The purpose of process groups is to allow related processes to be sent signals all at once (via killpg or kill). Such signaling is commonly done by the tty manager to interrupt or stop all components of a pipeline. The process group associated with a terminal can be set with the ioctl system call (if supported by the driver in use).

DIAGNOSTICS

getpgrp always returns a process group ID. setpgrp returns 0 if the process group was set, or -1 if not.

If the call fails **errno** will contain the following:

ESRCH	The process specified by **pid** does not exist

SEE ALSO

System Calls - `kill(), killpg(), setpgid(), setsid()`

Library Functions - `tcgetpgrp(), tcsetpgrp()`

NAME

`getpid`, `getppid` – get process identification number

SYNOPSIS

```
pid_t getpid()

pid_t getppid()
```

DESCRIPTION

`getpid` returns the process ID of the calling process. `getppid` returns the process ID of the parent of the calling process. All process IDs are unique at any one instant, although they will be re-used over time.

NAME

`getprio` – get process priority

SYNOPSIS

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

int getprio (pid)
pid_t pid;
```

DESCRIPTION

`getprio` returns the priority of the process specified by **pid**. If **pid** is 0, the priority for the calling process shall be returned.

DIAGNOSTICS

`getprio` returns the priority value for the process **pid** if successful, or -1 if not.

If `getprio` fails, **errno** will contain one of the following:

`ESRCH` **pid** does not specify a valid process.

CONFORMANCE

`getprio` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`getpid(), getscheduler(), setprio(), setscheduler()`
Example -	`getprio.c`

NAME

`getpriority, setpriority` – get or set the process

SYNOPSIS

```
#include <time.h>
#include <resource.h>

int getpriority(which, pid)
int which;
int pid;

int setpriority(which, pid, prio)
int which;
int pid;
int prio;
```

DESCRIPTION

When **which** is the constant `PRIO_PROCESS`, `getpriority` returns the priority of process **pid**. If **pid** is zero the priority of the calling process is returned. If the target process contains more than one thread of execution, one of the thread priorities is returned at random.

When **which** is the constant `PRIO_PROCESS`, `setpriority` sets the priority of process **pid** to **prio**. If **pid** is zero the priority of the calling process is set. If the target process contains more than one thread, then all the threads within the target process have their priority set to **prio**.

When **which** is `PRIO_PRIV`, then `setpriority` establishes **prio** as the highest priority that any non-privileged process can use in any future call to `setpriority` (privileged processes can always set any process to any priority). Only a privileged process can use `PRIO_PRIV` with `setpriority`.

If **which** is `PRIO_PRIV` in a call to `getpriority`, the current maximum non-privileged priority level is returned. Any process may make this inquiry.

A priority is a number from 0 (lowest) to 255 (highest). The thread with the highest priority that is not waiting on a system resource or I/O is by definition the current process.

pid may be a combined process/thread ID built using `BUILDPID`, in which case the indicated thread is used instead of the indicated process. By making **pid** refer to an individual thread, thread priorities within a single process can be varied and queried.

DIAGNOSTICS

If `getpriority` succeeds, it returns the priority. Otherwise it returns -1 and **errno** is set to one of the following values:

`EINVAL`	**which** is neither `PRIO_PROCESS` nor `PRIO_PRIV`.
`ESRCH`	Process/thread **pid** does not exist.

If `setpriority` succeeds, and **which** is equal to `PRIO_PRIV`, it returns the old maximum non-privileged priority. If `setpriority` succeeds, and **which** is equal to `PRIO_PROCESS`, it returns the old priority. Otherwise `setpriority` returns -1 and **errno** is set to one of the following values:

`EPERM`	**which** is `PRIO_PRIV` and the caller is not privileged.
`EINVAL`	**prio** is out of range, or which is neither `PRIO_PROCESS` nor `PRIO_PRIV`.
`EPERM`	Process/thread **pid** has a different user ID than the caller, and the caller is not privileged.
`EPERM`	The effective or real user IDs for process **pid** do not match the effective or real user IDs for the calling process, and the caller is not privileged.
`EPERM`	**prio** is greater than the maximum non-privileged priority, and the caller is not privileged.
`ESRCH`	Process/thread **pid** does not exist.

SEE ALSO

Library Functions - `BUILDPID()`

NAME

`getquantum, setquantum` – get or set the process quantum

SYNOPSIS

```
#include <types.h>
#include <sched.h>

int getquantum(which, tv, what)
int which;
struct timeval *tv;
int what;

int setquantum(which, tv, what)
int which;
struct timeval *tv;
int what;
```

DESCRIPTION

The quantum value specifies the duration of time that a task may have the cpu before it is forcibly removed. The task may voluntarily give up the cpu before its quantum expires, or it may be preempted by a higher priority process.

LynxOS provides facilities for manipulating the quantum value on a per-priority basis. Each priority level may have a different quantum value. Tasks that run under the `SCHED_DEFAULT` scheduler will have the quantum value that corresponds to their priority.

When **which** is the constant `DEFAULT_QUANTUM`, `getquantum` returns the quantum for priority **what**.

When **which** is the constant `DEFAULT_QUANTUM`, `setquantum` sets the quantum of priority **what** to **tv**. The **timeval** structure is defined as follows:

```
struct timeval {
  time_t tv_sec; /* seconds */
  time_t tv_usec; /* microseconds */
};
```

The caller must be privileged to use `setquantum`. The quantum is rounded up to the nearest clock tick, which usually has a resolution of hundredths of a second.

DIAGNOSTICS

If `getquantum` succeeds, it returns 0. Otherwise it returns -1 and **errno** is set to one of the following values:

EFAULT	**tv** does not point to a valid structure in the user's address space.
EINVAL	**which** is not DEFAULT_QUANTUM.
EINVAL	**which** is DEFAULT_QUANTUM, and **what** is not a valid priority.

If `setquantum` succeeds, it returns the old quantum value. Otherwise `setquantum` returns `-1` and **errno** is set to one of the following values:

EPERM	The caller is not privileged.
EFAULT	**tv** does not point to a valid structure in the user's address space.
EINVAL	**which** is not DEFAULT_QUANTUM.
EINVAL	**which** is DEFAULT_QUANTUM, and **what** is not a valid priority.
EINVAL	The time requested is less than or equal to 0.

SEE ALSO

System Calls - `getpriority(), getscheduler(), setscheduler()`

NAME

getrlimit, setrlimit – get or set a resource maximum

SYNOPSIS

```
#include <time.h>
#include <resource.h>

getrlimit(which, limit)
int which;
struct rlimit *limit;

setrlimit(which, limit)
int which;
struct rlimit *limit;
```

DESCRIPTION

`getrlimit` writes into **limit** a selected resource limit in effect for the calling process. `setrlimit` sets a selected limit to **limit**.

In both calls, the argument **which** selects one of the following resource limits:

RLIMIT_DATA	The maximum size of the process data segment. The data segment of a process changes upon calls to `brk` or `sbrk`.
RLIMIT_STACK	The maximum size of the process user stack segment. The user stack segment will automatically grow (if needed) up to this maximum size.
RLIMIT_CORE	The maximum size of a core dump of the process.
RLIMIT_THREAD	The stack “bottom” (highest address) and maximum length of the stack in pages. The stack “bottom” is returned in `limit->rlim_cur`; the maximum length is returned in `limit->rlim_max`. Since the stack grows downwards (towards lower addresses), the stack length given is a negative number of bytes.

A resource limit consists of two values, a current value and a maximum value. Non-privileged processes (those with non-zero effective user ID) can set the current value freely between zero and the maximum, and can set the maximum down. Only privileged processes can increase a maximum limit.

The thread stack bottom and limit cannot be changed at all. In the event of a stack overflow, an additional page of memory will be allocated to the stack of the offending thread for the duration of the signal handling (`SIGSEGV` is generated). Upon return from the signal handler, the extra page is de-allocated. If the extra page is overrun, the thread is killed with `SIGKILL`, resulting in the entire process' demise.

The structure for a resource limit is:

```
struct rlimit {
  long rlim_cur; /* current limit */
  long rlim_max; /* maximum limit */
};
```

If the maximum stack size limit is exceeded, the process will be killed with a `SIGKILL` signal. If the maximum data segment limit is exceeded, `brk` or `sbrk` calls will no longer be allowed to expand the data segment size.

DIAGNOSTICS

Both `getrlimit` and `setrlimit` return 0 if successful, or -1 if not.

If the call fails **`errno`** will contain one of the following:

`EFAULT`	**`limit`** does not point to allocated memory in the process' address space.
`EINVAL`	The requested current limit is greater than the requested maximum limit, or one or both of the requested limits is negative.
`EPERM`	A non-privileged process attempted to set one of the limits beyond the current maximum limit.

NOTE

The operating system allows the limits to be set to values beyond the physical capacity of the system.

NAME

getrusage – get resource use

SYNOPSIS

```
#include <time.h>
#include <resource.h>

getrusage(who, usage)
int who;
struct rusage *usage;
```

DESCRIPTION

getrusage gets the resource use for the calling process and/or all of its terminated child processes. The argument **who** should be RUSAGE_SELF for the calling process, RUSAGE_CHILDREN for the child processes, or RUSAGE_SUM for the sum of the calling process and the child processes.

The resource information is written to the structure pointed to by **usage**. The structure is defined (in resource.h) as follows:

```
struct rusage {
  struct timeval   ru_utime;  /* total user time */
  struct timeval   ru_stime;  /* total system time */
  int   ru_maxrss;   /* unused by LynxOS */
  int   ru_ixrss;   /* unused by LynxOS */
  int   ru_idrss;   /* unused by LynxOS */
  int   ru_isrss;   /* unused by LynxOS */
  int   ru_minflt;   /* unused by LynxOS */
  int   ru_majflt;   /* unused by LynxOS */
  int   ru_nswap;   /* unused by LynxOS */
  int   ru_inblock;   /* unused by LynxOS */
  int   ru_oublock;   /* unused by LynxOS */
  int   ru_msgsnd;   /* unused by LynxOS */
  int   ru_msgrcv;   /* unused by LynxOS */
  int   ru_nsignals;   /* unused by LynxOS */
  int   ru_nvcsw;   /* unused by LynxOS */
  int   ru_nivcsw;  /* unused by LynxOS */
};
```

This structure exists for BSD compatibility and only the `ru_utime` and `ru_stime` fields are used by LynxOS. The other fields are unmaintained by LynxOS.

DIAGNOSTICS

`getrusage` returns 0 if successful, or -1 if not.

If the call fails **errno** will contain one of the following:

EFAULT	**usage** does not point to allocated memory in the process' address space.
EINVAL	**who** is not one of RUSAGE_SELF, RUSAGE_CHILDREN, or RUSAGE_SUM.

SEE ALSO

System Calls - `gettimeofday()`

NAME

`getscheduler` – return process scheduling policy

SYNOPSIS

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

int getscheduler(pid)
pid_t pid;
```

DESCRIPTION

`getscheduler` returns the scheduling policy of the process specified by **`pid`**. If **`pid`** is `0`, the priority and scheduling policy shall be returned for the calling process.

DIAGNOSTICS

`getscheduler` returns the current scheduling policy if successful, or `–1` if not.

If `getscheduler` fails, **`errno`** will contain one of the following:

`ESRCH`	**`pid`** does not specify a valid process.

CONFORMANCE

`getscheduler` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`getpid()`, `getprio()`, `setprio()`, `setscheduler()`
Example -	`getscheduler.c`

NAME

getsockname – get socket name

SYNOPSIS

```
getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

DESCRIPTION

getsockname returns the current **name** for the specified socket. The **namelen** parameter should be initialized to indicate the amount of space pointed to by **name**. On return it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

getsockname returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain one of the following:

EBADF	The argument **s** is not a valid descriptor.
ENOTSOCK	The argument **s** is a file, not a socket.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
EFAULT	The **name** parameter points to memory not in a valid part of the process address space.

SEE ALSO

System Calls - bind(), socket()

NAME

getsockopt,
setsockopt – get and set options on sockets

SYNOPSIS

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

int getsockopt(int *s*, **int** *level*, **int** *optname*, **void** **optval*, **int** **optlen*);

int setsockopt(int *s*, **int** *level*, **int** *optname*, **const void** **optval*, **int** *optlen*);

DESCRIPTION

getsockopt() and setsockopt() manipulate the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see getprotoent(3).

The parameters *optval* and *optlen* are used to access option values for setsockopt(). For getsockopt() they identify a buffer in which the value for the requested option(s) are to be returned. For getsockopt(), *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, optval may be NULL.

optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file `<sys/socket.h>` contains definitions for socket level options, described below. Options at other protocol levels vary in format and name; consult the appropriate entries in section 4 of the manual.

Most socket-level options utilize an `int` parameter for *optval*. For `setsockopt()`, the parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. `SO_LINGER` uses a `struct linger` parameter, defined in `<sys/socket.h>`, which specifies the desired state of the option and the linger interval (see below). `SO_SNDTIMEO` and `SO_RCVTIMEO` use a `struct timeval` parameter, defined in `<sys/time.h>`.

OPTIONS

The following options are recognized at the socket level. Except as noted, each may be examined with `getsockopt()` and set with `setsockopt()`.

`SO_DEBUG`	enables recording of debugging information
`SO_REUSEADDR`	enables local address reuse
`SO_REUSEPORT`	enables duplicate address and port bindings
`SO_KEEPALIVE`	enables keep connections alive
`SO_DONTROUTE`	enables routing bypass for outgoing messages
`SO_LINGER`	linger on close if data present
`SO_BROADCAST`	enables permission to transmit broadcast messages
`SO_OOBINLINE`	enables reception of out-of-band data in band
`SO_SNDBUF`	set buffer size for output
`SO_RCVBUF`	set buffer size for input
`SO_SNDLOWAT`	set minimum count for output
`SO_RCVLOWAT`	set minimum count for input
`SO_SNDTIMEO`	set timeout value for output
`SO_RCVTIMEO`	set timeout value for input

`SO_TYPE`	get the type of the socket (get only)
`SO_ERROR`	get and clear error on the socket (get only)

`SO_DEBUG` enables debugging in the underlying protocol modules. `SO_REUSEADDR` indicates that the rules used in validating addresses supplied in a `bind`(2) call should allow reuse of local addresses.

`SO_REUSEPORT` allows completely duplicate bindings by multiple processes if they all set `SO_REUSEPORT` before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. `SO_KEEPALIVE` enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a `SIGPIPE` signal when attempting to send data.

`SO_DONTROUTE` indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

`SO_LINGER` controls the action taken when unsent messages are queued on socket and a `close()` is performed. If the socket promises reliable delivery of data and `SO_LINGER` is set, the system will block the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the `setsockopt()` call when `SO_LINGER` is requested). If `SO_LINGER` is disabled and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option `SO_BROADCAST` requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the `SO_OOBINLINE` option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv or read calls without the `MSG_OOB` flag. Some protocols always behave as if this option is set. `SO_SNDBUF` and `SO_RCVBUF` are options to adjust the normal buffer sizes allocated for output and input buffers, respectively.

The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

`SO_SNDLOWAT` is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A `select()` operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for `SO_SNDLOWAT` is set to a convenient size for network efficiency, often 1024. `SO_RCVLOWAT` is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for `SO_RCVLOWAT` is 1. If `SO_RCVLOWAT` is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned.

`SO_SNDTIMEO` is an option to set a timeout value for output operations. It accepts a `struct timeval` parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or with the error `EWOULDBLOCK` if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output. `SO_RCVTIMEO` is an option to set a timeout value for input operations. It accepts a `struct timeval` parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error `EWOULDBLOCK` if no data were received.

Finally, `SO_TYPE` and `SO_ERROR` are options used only with `getsockopt()`. `SO_TYPE` returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. `SO_ERROR` returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUES

A 0 is returned if the call succeeds, –1 if it fails.

ERRORS

The call succeeds unless:

`[EBADF]`	The argument s is not a valid descriptor.
`[ENOTSOCK]`	The argument s is a file, not a socket.
`[ENOPROTOOPT]`	The option is unknown at the level indicated.
`[EFAULT]`	The address pointed to by optval is not in a valid part of the process address space. For `getsockopt()`, this error may also be returned if optlen is not in a valid part of the process address space.

SEE ALSO

System Calls -	`ioctl()`, `socket()`
Library Functions -	`getprotoent()`
File Formats -	`protocols`

BUGS

Several of the socket options should be handled at lower levels of the system.

HISTORY

The getsockopt system call appeared in 4.2BSD.

NAME

`_getstid` – get system thread ID

SYNOPSIS

`tid_t _getstid()`

DESCRIPTION

`getstid` returns the system thread ID for the calling system thread. The thread ID is used as a thread identifier in most thread control system and library calls.

DIAGNOSTICS

`_getstid` never fails.

NOTE

`_getstid` is functionally equivalent to the `getstid` library routine, the only difference being that `_getstid` is a kernel call and `getstid` is nothing more than a macro accessing a piece of thread-specific data which is initialized to contain the thread ID when the thread is first started up. This thread-specific value can be accidentally overwritten by an errant program, just as any value in the writable address space of the thread can be.

Correct programs using the C library do not need to use `_getstid`.

SEE ALSO

System Calls - `st_build(), st_join(), fork()`

Library Functions - `getstid(), stattr_default(), st_new()`

NAME

gettimeofday,
settimeofday – get or set the system time and date

SYNOPSIS

```
#include <time.h>

gettimeofday(time, zone)
struct timeval *time;
struct timezone *zone;

settimeofday(time, zone)
struct timeval *time;
struct timezone *zone;
```

DESCRIPTION

`gettimeofday` writes the current time to the **`timeval`** structure pointed to by **`time`**, and the time zone to the **`timezone`** structure pointed to by **`zone`**. `settimeofday` sets the current time from the given structures.

The **`time`** and **`timezone`** structures are defined (in <`time.h`>) as follows:

```
struct timeval {
    time_t tv_sec;      /* time in seconds */
    time_t tv_usec;     /* microseconds    */
};
struct timezone {
    int tz_minuteswest;  /* Minutes west of Greenwich */
    int tz_dsttime;      /* Type of dst correction    */
};
```

The time is given as the number of seconds and microseconds since the start of LynxOS time, usually midnight GMT on 1 January 1970. The time zone is given as the number of minutes west of Greenwich and the type of daylight savings time correction to apply when appropriate.

`settimeofday` can only be called by processes whose effective user ID is 0 (super-user processes).

DIAGNOSTICS

`gettimeofday` and `settimeofday` return 0 if successful, or -1 if not. If the call fails, **`errno`** will contain one of the following:

`EFAULT`	**`time`** or **`zone`** do not point to allocated memory in the process' address space.
`EPERM`	The effective user ID of the calling process is not 0.

SEE ALSO

Utility Programs - `date`

Library Functions - `ctime()`

NAME

`getuid, geteuid` – get current user ID

SYNOPSIS

`int getuid()`

`int geteuid()`

DESCRIPTION

The routines `getuid` and `geteuid` return the real or effective user ID number.

The real user ID can only be changed by the `setreuid` system call. This is generally done by the login program before starting a new shell or application. The effective user ID can be set by `setreuid` but is more often changed when executing a program with the `S_ISUID` ("set user ID") bit enabled. Such programs can use `getuid` to determine the original user ID, and `geteuid` to compute access permissions.

SEE ALSO

System Calls - `setreuid(), stat()`

NAME

`info` – get information from the current operating system

SYNOPSIS

```
#include <info.h>

long info(what)
int what;
```

DESCRIPTION

`info` returns a value used within the operating system corresponding to the given value of **`what`**. Values returned by `info` come in three types: quantities, direct addresses, and indirect addresses. The following is a list of the various values for **`what`**, as defined in the header file `<info.h>`, along with a description of the corresponding value returned by `info`:

`_I_PROCTAB`	(direct address) Address of the process table.
`_I_USEMTABLE`	(direct address) Address of the user semaphore table.
`_I_SMEMTABLE`	(direct address) Address of the shared memory table.
`_I_DRIVERS`	(direct address) Address of the driver table.
`_I_CDEVSW`	(direct address) Address of the character device table.
`_I_BDEVSW`	(direct address) Address of the block device table.
`_I_NPROCTAB`	(quantity) The size of the process table.
`_I_NUSEMTABLE`	(quantity) The size of the user semaphore table.
`_I_NSMEMTABLE`	(quantity) The size of the shared memory table.
`_I_NDRIVERS`	(quantity) The size of the driver table.
`_I_NCDEVSW`	(quantity) The size of the character device table.
`_I_NBDEVSW`	(quantity) The size of the block device table.
`_I_FREEPAGES`	(direct address) Address of the variable containing the number of free pages in the system.

`_I_STASKTAB`	(direct address) Address of the system thread table.
`_I_NSTASKTAB`	(quantity) Size of the system thread table.
`_I_INODETABLE`	(indirect address) Address of the pointer to the beginning of the in-core inode table.
`_I_NUM_INODES`	(quantity) Size of the in-core inode table.
`_I_MSGTAB`	(direct address) Address of System V compatible message queue table.
`_I_NMSGTAB`	(quantity) Size of System V compatible message queue table.
`_I_VSEMTAB`	(direct address) Address of System V compatible semaphore table.
`_I_NVSEMTAB`	(quantity) Size of System V compatible semaphore table.
`_I_VSHMTAB`	(direct address) Address of System V compatible shared memory table.
`_I_NVSHMTAB`	(quantity) Size of System V compatible shared memory table.
`_I_EXECARGLEN`	(quantity) Maximum number of characters in the exec system call's arguments plus environment.
`_I_HZ`	(quantity) The value of the constant HZ.
`_I_NGROUPS`	(quantity) Maximum number of access groups for a process.
`_I_USR_NFDS`	(quantity) Maximum number of open files per process.
`_I_LMAX_LINKS`	(quantity) Maximum number of hard links per file.
`_I_LINELEN`	(quantity) Maximum number on `tty` input line.
`_I_QSIZE`	(quantity) Maximum number on `tty` input queue.
`_I_MAXNAMLEN`	(quantity) Maximum number of characters in a file name.
`_I_MAXPATHLEN`	(quantity) Maximum number of characters in a path name.

`_I_PIPESIZE`	(quantity) Size of a pipe.
`_I_FILE_TABLE`	(indirect address) Address of the pointer to the beginning of the file table.
`_I_FILE_FREE`	(indirect address) Address of the pointer to the beginning of the free file linked list.
`_I_NFILES`	(quantity) Size of the file table.
`_I_MOUNTTABLE`	(indirect address) Address of the pointer to the beginning of the mount table.
`_I_NMOUNTS`	(quantity) Size of the mount table.
`_I_NTHREADS`	(quantity) Number of system threads.
`_I_PHYSBASE`	(direct address) Virtual address where physical memory begins.
`_I_TCPIP_INFO`	(direct address) Address of `tcpip_info` table.
`_I_FREEVPAGES`	(direct address) Address of the variable containing the number of free virtual memory pages.
`_I_VMEM`	(direct address) Address of the flag variable indicating whether or not virtual memory has been engaged.
`_I_VMSTAT`	(direct address) Address of the virtual memory statistics structure, `struct vm_statistics`, which is defined in `<machine/mem.h>`.

A direct address returned by `info` can be used to access the actual operating system table by opening the memory pseudo-device (usually called `/dev/mem`), seeking to the address with `lseek`, and reading memory with `read`. An indirect address requires an extra step: seek to the indirect address with `lseek`; read the 4-byte direct address using read; seek to this direct address using `lseek`; and finally read the memory with `read`.

SEE ALSO

System Calls - `open(), read(), lseek()`

NAME

`ioctl` – I/O device control

SYNOPSIS

```
#include <ioctl.h>

ioctl(fd, request, arg)
int fd;
int request;
char *arg;
```

DESCRIPTION

`ioctl` provides a way of sending special commands to a device driver or a stream. The call sends the value of **`request`** and the pointer **`arg`** to the driver for the device or stream associated with the file descriptor **`fd.`** Different device drivers support different `ioctl` calls; consult the relevant device interface documentation. There are special `ioctl` calls for STREAMS; consult the `streamio` manual page in the Device Facilities section of *LynxOS Reference Manuals*.

Normally, an `ioctl` call affects only the minor device open for the descriptor. Individual drivers may, however, support `ioctl` commands which have effect over the entire major device.

DIAGNOSTICS

`ioctl` returns 0 if successful, or -1 if not.
If the call fails, **`errno`** will contain one of the following:

`EBADF`	**`fd`** is not a valid file descriptor.
`EFAULT`	**`arg`** `does not point to allocated memory in the processes' address space`
`EFTYPE`	**`fd`** refers to a POSIX `ipc` special file.
`EINTR`	A signal interrupted the `ioctl` call.
`EINVAL`	**`request`** or **`arg`** is invalid.
`ENODEV`	**`fd`** does not refer to a device.

`ENXIO`	`request` and `arg` are legal but the sub-device cannot perform services specified by **`request`**.

SEE ALSO

System Calls - `fcntl()`

NAME

kill – send a signal

SYNOPSIS

```
#include <signal.h>

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

DESCRIPTION

kill sends the signal **sig** to the process number **pid**. The signal number must lie between 1 and 32 (inclusive). If signal 0 is sent, the call succeeds (returns 0), if the process being signalled exists; no signal is sent in this case.

If **pid** is 0, the signal is sent to all other processes in the process group of the caller. If **pid** is -1, the signal is sent to all processes on the system except processes 0 and 1.

Otherwise, the **pid** is interpreted as a concatenation of a process ID and a thread ID built using BUILDPID (process_id, thread_id). If the thread_id part of **pid** is zero, then **pid** refers to the entire process named by the process_id part of **pid**. If the process_id part of **pid** is zero, then the thread_id is interpreted within the current process.

A process can send a signal to itself.

DIAGNOSTICS

kill returns 0 if successful, or -1 if not. If the call fails **errno** will contain one of the following:

ESRCH	Process and/or thread indicated by **pid** does not exist.
EINVAL	**sig** is invalid.

SEE ALSO

System Calls - `getpid()`, `getpgrp()`, `killpg()`, `sigvec()`, `ekill()`

Library Functions - `getstid()`, `BUILDPID()`

NAME

`killpg` – send a signal to a process group

SYNOPSIS

```
#include <signal.h>

killpg(pgrp, sig)
int pgrp;
int sig;
```

DESCRIPTION

`killpg` sends the signal **`sig`** to the process group **`pgrp`**. The signal number must lie between 1 and 32 (inclusive).

DIAGNOSTICS

`killpg` returns `0` if successful, or `-1` if not. If the call fails **`errno`** will contain one of the following:

`EINVAL`	**`sig`** is invalid.
`ESRCH`	No process can be found in the process group specified by **`pgrp`**.

SEE ALSO

System Calls - `getpgrp(), kill(), sigvec()`

NAME

link – make a hard link to file

SYNOPSIS

```
link(path1, path2)
char *path1;
char *path2;
```

DESCRIPTION

link adds a directory entry named **path2** which refers to the inode referenced by directory entry **path1**. Thereafter, any references through **path2** will access the same actual file as those through **path1**.

Both **path1** and **path2** must be on the same file system. Both share identical access rights to the file, because all such information is kept in the inode.

A hard link to a directory is not allowed.

path1 must exist; **path2** must not exist.

DIAGNOSTICS

Link returns 0 if the link was made, or -1 if not. If the call fails, **errno** will contain one of the following:

ENOTDIR	An element of either path prefix is not a directory.
ENAMETOOLONG	**path1** or **path2** is too long.
ENOENT	An element of either path prefix does not exist.
ENOENT	**path1** does not exist.
EEXIST	**path2** does exist.
EPERM	**path1** refers to a directory.
ELOOP	Too many symbolic links were encountered when traversing one of the paths.

EROFS	Making the link requires writing to a directory on a read-only file system.
EACCES	Making the link requires writing to a directory that denies write permission.
EFAULT	**path1** or **path2** does not point to allocated memory in the process' address space.
EXDEV	**path1** and **path2** are on different file systems.

SEE ALSO

System Calls - `symlink(), unlink()`

NAME

`listen` – listen for connections on a socket

SYNOPSIS

```
listen(s, backlog)
int s, backlog;
```

DESCRIPTION

To accept connections, a socket is first created with `socket`, a **backlog** for incoming connections is specified with `listen` and then the connections are accepted with `accept`. The `listen` call applies only to sockets of type `SOCK_STREAM`.

The **backlog** parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of `ECONNREFUSED`; or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

DIAGNOSTICS

`listen` returns `0` if successful, or `–1` if not.

If the call fails, **errno** will contain one of the following:

`EBADF`	The argument **s** is not a valid descriptor.
`ENOTSOCK`	The argument **s** is not a socket.
`EOPNOTSUPP`	The socket is not of a type that supports the operation `listen`.

NOTE

The backlog is ignored (silently).

SEE ALSO

System Calls - `accept(), connect(), socket()`

NAME

`lockf` – advisory record locking on files

SYNOPSIS

```
#include <unistd.h>

lockf(fd, func, size)
int fd, func;
long size;
```

DESCRIPTION

`lockf` allows regions of a file to be selectively locked and unlocked in an advisory sense. Other processes which attempt to lock an already locked portion of a file will fail and may elect to abort the operation or block until the lock becomes possible. The file remains completely accessible to processes that do not attempt locks.

The file involved is specified by the file descriptor **`fd`**. The operation performed by `lockf` is given by **`func`**, which may be any of the following values (defined in the include file **`unistd.h`**).

`F_ULOCK`	Unlock a previously locked region.
`F_LOCK`	Lock a region for exclusive use.
`F_TLOCK`	Test and lock a region for exclusive use.
`F_TEST`	Test a region for other locks.

`F_TEST` is used to detect if a lock by another process exists on the specified region. `F_LOCK` and `F_TLOCK` both lock a region of a file if the region is available. If the region is already locked, `F_LOCK` causes the current process to wait until the lock is removed while `F_TLOCK` causes an error (`EACCES`) to be returned. `F_ULOCK` removes any locks owned by the current process from a region of the file. Locks are also removed when the file is closed, or when the owning process terminates.

The **`size`** argument gives the number of contiguous bytes to be locked or unlocked. If **`size`** is greater than zero, the affected region ranges from the current offset forward. If **`size`** is negative, the region ranges from **`size`** bytes before the current offset up to the byte just before the current offset. If **size** is zero the region runs from the current offset through the largest file offset, and later extends automatically when the file grows. An area need not be allocated to the file in order to be locked; for example, locks may exist past the end of the file.

Processes waiting for locks are interrupted by signals. `setitimer` may thus be used to provide a timeout facility.

A succession of locks may establish overlapping locked regions. The system merges overlapping locks into single locks. Likewise, portions of locked regions may be unlocked without effect on locks on remaining portions. If at any time a `lock` or `unlock` request requires that a new region be added, and no room in the internal lock table is available, the operation will fail and an error is returned.

A potential for deadlock occurs if a process controlling a locked region is caused to wait by attempting to lock a locked region of another process. Therefore calls to `lockf` scan for a deadlock prior to waiting for a locked resource. An error is returned if waiting for the locked resource would cause deadlock.

NOTE

Interaction between locks and buffered input/output systems may produce anomalous situations. Careful (even aggressive) use of buffer flushing requests is recommended.

DIAGNOSTICS

`lockf` returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain one of the following:

EBADF	**fd** is not a valid file descriptor.
EACCES	**func** is F_TLOCK or F_TEST, and the requested lock would conflict with other locks.
EDEADLK	**func** is F_LOCK or F_TLOCK and the requested lock would cause a deadlock to occur.

EDEADLK	**func** is F_ULOCK and there are no more entries in the lock table.
ENOLCK	**func** is F_LOCK or F_TLOCK and there are no more entries in the lock table.
EINVAL	**func** is not one of F_ULOCK, F_LOCK, F_TLOCK, or F_TEST.

NAME

`lseek` – move read/write file pointer

SYNOPSIS

```
long lseek(fd, offset, whence)
int fd;
long offset;
int whence;
```

DESCRIPTION

`lseek` sets the seek pointer associated with file descriptor **fd** a distance of **offset** bytes from the start of the file, the current position, or the end of the file depending on whether **whence** is 0, 1, or 2.

DIAGNOSTICS

If successful, `lseek` returns the pointer position measured in bytes from the beginning of the file.

If the seek pointer could not be set, `lseek` returns -1.

If the call fails, **errno** will contain one of the following:

EBADF	**fd** is not an open file descriptor.
EFTYPE	**fd** refers to a POSIX ipc special file.
ESPIPE	**fd** refers to a pipe.
EINVAL	**whence** is not 0, 1 or 2.
EINVAL	The resulting pointer position would be negative.

NAME

memlk,memunlk – process memory locking

SYNOPSIS

```
#include <memlk.h>

int memlk(type,addr,size)
int type ;
void *addr ;
size_t size ;

int memunlk(type,addr,size)
int type ;
void *addr ;
size_t size ;
```

DESCRIPTION

The `memlk()` function allows the calling process to make resident the region described by the input argument. The locked region is made resident and remains resident until unlocked.

The `memunlk()` function allows the calling process to unlock the region referenced by the input arguments, that has been previously locked by `memlk()`.

type specifies one of the following memory region types as defined by the `<memlk.h>` header file.

PROLOCK	The entire address space of the process.
DATLOCK	The data segment of the process.
TXTLOCK	The text segment of the process.
STKLOCK	The stack segment of the process.
REGLOCK	The region corresponding to the specified range of addresses.

addr specifies the starting address of the region. This argument is used only if the **type** is REGLOCK.

size specifies the size, in bytes, of the region. This argument is used only if the **type** is REGLOCK.

Upon successful completion, the memlk() function returns a value zero. Otherwise, none of the memory is locked, a value of -1 is returned and **errno** is set to indicate the error.

Upon successful completion, the memunlk() function returns a value zero. Otherwise, none of the memory is unlocked, a value of -1 is returned and **errno** is set to indicate the error.

NOTE

Multiple locks can be placed on a memory region. Each region locked by a memlk() call has to be unlocked by a corresponding memunlk() function call. The release of the last lock on the region allows it to become non-resident.

DIAGNOSTICS

If any of the following conditions occur, the memlk() or memunlk() calls return a value of –1 and **errno** is set to:

EINVAL	The **type** argument is invalid. For memunlk(), the memory region is not locked.
ENOMEM	The calling process has exceeded system defined limits (includes too many regions locked).
EFAULT	The **addr** argument is not a valid address.

CONFORMANCE

memlk and memunlk are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Example - memlk.c, memunlk.c

NAME

`mkcontig` – make a contiguous file

SYNOPSIS

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

int mkcontig(path, mode, length)
char *path;
int mode;
long length;
```

DESCRIPTION

`mkcontig` creates a contiguous file named **path** with size **length** in bytes. mkcontig will then divide the length by 512 and round down. This is the actual size, in blocks, of the contiguous file. If the file already exists, it must be a contiguous file of 0 block size. Upon successful completion, permission for the new file is set to the value obtained by bitwise-AND of **mode** and the complement of current `umask`.

DIAGNOSTICS

`mkcontig` returns 0 if successful, or -1 if not.

If `mkcontig` fails, **errno** will contain one of the following:

EBUSY	Some process has this file open.
ENOTDIR	An element of the path prefix is not a directory.
EISDIR	The file is a directory, and write access was requested.
EMFILE	The maximum number of files open per process has been reached.
ENAMETOOLONG	The path name is too long.
ENOENT	An element of the path prefix does not exist.
ENXIO	The file is a device special file.

`ENXIO`	`O_NDELAY` was requested, and the file is busy.
`EEXIST`	`O_EXCL` was requested, and the file exists.
`ELOOP`	Too many symbolic links were encountered when traversing the path.
`EROFS`	Write access is denied because file is on a read- only file system.
`ETXTBSY`	Write access is denied because the file is being executed.
`EACCES`	Requested access for a file denied according to the mode of the file, or a directory in the path denied search permission.
`EFAULT`	**`path`** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - `open()`

NAME

`mkdir` – make a directory

SYNOPSIS

```
mkdir(path, mode)
char *path;
int mode;
```

DESCRIPTION

`mkdir` creates a new directory named **`path`** containing only the "**`.`**" and "**`..`**" entries. The **`mode`** for the new directory is given by **`mode`**, modified by the mode mask of the calling process.

DIAGNOSTICS

`mkdir` returns 0 if the directory was created, or -1 if not.

If the call fails, **`errno`** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	An element of the path prefix does not exist.
EEXIST	The named file already exists.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	A directory in the path denied search permission, or the parent directory of the directory to be created denied write permission.
EFAULT	**`path`** does not point to allocated memory in the process' address space.
EIO	An I/O error occurred.
EROFS	The directory would reside on a read-only file system.
ENOSPC	The file system is out of space.

SEE ALSO

System Calls - `chmod(), stat(), umask()`

NAME

`mkmq` – create a new message queue special file

SYNOPSIS

```
#include <sys/mqueue.h>

int mkmq(path, mode)
char *path;
mode_t mode;
```

DESCRIPTION

`mkmq` creates a new message queue special file named by the pathname pointed to by **path**.

The **mode** of the new message queue file is initialized from **mode**. If `MQ_PERSIST` is specified in **mode**, then a persistent message queue special file is created; otherwise a non-persistent message queue special file is created. The protection part of the **mode** parameter is modified by the process' file creation mask (see `umask`). When bits in **mode** other than the protection part and the `MQ_PERSIST` are specified in the **mode** parameter, incorrect permissions may result for the created file.

The difference between a persistent and non-persistent message queue special file is noticed in the actions of the `open` and `close` system calls. This information is summarized here for quick reference:

If the created file is persistent, then any unreceived messages in the queue will remain in the file after the last `close` of the file. On the first `open` of a persistent message queue special file, any unreceived messages left there will be available. The state of the file and its contents will be unchanged until the file is removed or until system reboot.

If the created file is not persistent, then after the last `close` of the file, the pathname remains valid, but the contents of the file are undefined. A non-persistent file is always initialized to the empty state on the first `open`, removing any "leftover" messages. (See `open` and `close`.)

Note that on `close` of any message queue special file, any messages sent by the closing process with the `MSG_USE` flag are removed from the message queue. (The closing process' data buffer which was to be used by the receiver will no longer be available). (See `mqreceive`.)

The name of the message queue special file created by the `mkmq` function will remain a "valid" pathname until the pathname is removed or until system reboot. After a system reboot, the pathname and file state are invalid. The file will need to be created using `mkmq` again. The existence of an invalid pathname is ignored by `mkmq` when re-creating the file by the same name.

The message queue's owner ID and group ID will be set to the process' effective user ID, and effective group ID, respectively.

Upon successful completion, the `mkmq` function will update the `st_atime`, `st_ctime`, and `st_mtime` fields of the file (see `stat`). Also, the `st_ctime` and `st_mtime` fields of the directory that contain the new entry will be updated.

Use `mqsetattr` to change the default queue attributes.

DIAGNOSTICS

`mkmq` returns 0 when successful and -1 when unsuccessful. If the call fails, **errno** will contain one of the following:

EACCES	A component of the path prefix denies search permission.
EEXIST	The named file already exists.
EFTYPE	The **path** refers to a file system that does not support message queue special files.
ENAMETOOLONG	The length of the path string exceeds `MAXPATHLEN`.
ENOENT	A component of the path prefix does not exist or the **path** argument points to an empty string.
ENOSPC	The directory that would contain the new file cannot be extended or the file system is out of space.

`EROFS`	The directory in which the message queue special file is to be created resides on a read-only file system.
`ENOMEM`	Out of a system resource.
`EINVAL`	Illegal **`mode`**.
`EMFILE`	Too many open files.
`EFAULT`	The **`path`** argument does not point to allocated memory in the process's address space.
`ENOTDIR`	A component of the **`path`** prefix is not a directory.

CONFORMANCE

`mkmq` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `chmod(), stat(), umask(), mqgetattr(), mqsetattr(), open(), close(), mqreceive()`

The following calls operate on POSIX message queue special files:

System Calls - `mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()`

Library Functions - `mqgetpid()`

Example - `mkmq.c`

NAME

`mknod` – make a device special file or a named pipe

SYNOPSIS

```
#include <types.h>
#include <stat.h>

mknod(path, mode, dev)
char *path;
int mode;
int dev;
```

DESCRIPTION

`mknod` creates a new file named **`path`** that is either a device special file for device number **`dev`** or a named pipe (FIFO). The **`mode`** of the file is set to the low-order 9 bits of **`mode`**, modified by the file creation mode mask of the calling process.

`mode` also contains the file type bits. These bits must be `S_IFBLK` for a block device, `S_IFCHR` for a character device, or `S_IFFIFO` for a named pipe (FIFO). In the case of a named pipe, **`dev`** is ignored.

This system call checks the file type. If the file type isn't `IF_POSIX_IPC_NON_PERSIST`, `IF_POSIX_IPC_PERSIST`, or `IFFIFO` it is only available to processes running with effective user ID 0 (super-user processes). These special file types are for creating `IPC` channels.

NOTE

In addition to special files, `mknod` can also be used to make directories or regular files, although `mkdir` and `open` are more typically used for these purposes. For a directory, the file type bits must be set to `S_IFDIR`. For a regular file, they must be `S_IFREG`. In both cases, **`dev`** is ignored.

DIAGNOSTICS

`mknod` returns 0 if the file was created, or -1 if not.

If the call fails, **`errno`** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	An element of the path prefix does not exist.
EPERM	The effective user ID of the calling process is not 0.
ELOOP	Too many symbolic links were encountered when traversing the path.
EFAULT	**path** does not point to allocated memory in the process's address space.
EIO	An I/O error occurred.
EROFS	The file would reside on a read-only file system.
EEXIST	The named file already exists.
EINVAL	The file type bits of **mode** are not one of S_IFBLK, S_IFCHR, S_IFFIFO, S_IFDIR, or S_IFREG.

SEE ALSO

System Calls - `chmod(), stat(), umask()`

NAME

`mksem` – create a binary semaphore special file (POSIX.4, Draft 9)

SYNOPSIS

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

int mksem(path, mode, state)
char *path;
mode_t mode;
int stat;
```

DESCRIPTION

`mksem` creates a binary semaphore special file. The file is named by the pathname pointed to by **path**. The path cannot be to an NFS mounted file system. The **mode** of the special file is set to **mode** modified by the process' file creation mask (see `umask`). The length of the special file is 0. The file's owner-ID is set to the process' effective user ID, and the file's group ID is set to the group ID of the directory in which the file is being created.

If `SEM_PERSIST` is specified in **mode**, a persistent binary semaphore file is created. Otherwise, a non-persistent binary semaphore file is created. A persistent binary semaphore file maintains the **state** of the binary semaphore after a last close of the file. A subsequent `open` of a persistent binary semaphore file will find the binary semaphore in the same **state** as at the time of the final close. A non-persistent binary semaphore does not preserve the **state** after final close, and a subsequent `open` will find the non-persistent binary semaphore in the `SEM_UNLOCKED` state.

If `SEM_PERSIST` is specified in **mode**, the initial state of the binary semaphore is taken from **state** as long as it is not `SEM_PRIO_INH`. If **state** is `SEM_LOCKED`, the initial state of the semaphore is locked. If **state** is `SEM_UNLOCKED`, the initial state of the semaphore is unlocked. If `SEM_PERSIST` is not specified in **mode**, the initial state of the semaphore will be `SEM_UNLOCKED`.

If **state** is `SEM_PRIO_INH`, then priority inheritance will be performed on the semaphore. Briefly, priority inheritance means that any process owning the semaphore will run at a priority equal to the maximum of its own priority and the priorities of all processes currently blocked on the semaphore. A priority inheritance semaphore is always created in the unlocked **state**. Since priority inheritance presupposes an owner of the semaphore, it makes no sense to create a persistent binary sempahore that does priority inheritance. `mksem` will return an error in such a case.

The pathname **path** remains a valid binary semaphore special file until it is unlinked or the system is rebooted. The pathname becomes invalid and no **state** persists across a system reboot.

Note that the binary semaphore special file must be opened `O_RDWR`. Any other combination of flags will cause `open` to fail with **errno** set to `EINVAL`.

DIAGNOSTICS

`mksem` returns `0` if successful. Otherwise `mksem` returns `-1` and **errno** is set to one of the following values:

EACCES	A component of the path denies search permission.
EEXIST	The pathname **path** already exists.
EINVAL	**state** is not one of `SEM_LOCKED`, `SEM_UNLOCKED`, or `SEM_PRIO_INH`.
EINVAL	`SEM_PERSIST` is specified in **mode** and **state** is `SEM_PRIO_INH`.
EINVAL	A bit other than the **mode** bits or `SEM_PERSIST` is specified in **mode**.
EMFILE	Too many open files.
ENAMETOOLONG	The length of the **path** string exceeds `MAXPATHLEN`.
ENOENT	A component of **path** does not exist or **path** points to an empty string.
ENOSPC	The target directory cannot be extended or the file system is out of space.
EROFS	The file system is read-only.
ENOTDIR	A component of the pathname is not a directory.

CONFORMANCE

`mksem` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`close()`, `open()`, `semwait()`, `semifwait()`, `sempost()`, `semifpost()`
Example -	`mksem.c`

NAME

mkshm – create a new shared memory special file

SYNOPSIS

```
#include <sys/shmmap.h>

int mkshm(path, mode, size)
char *path;
mode_t mode;
off_t size;
```

DESCRIPTION

mkshm creates a new shared memory special file named by the pathname pointed to by **path**.

The **mode** of the new shared memory file is initialized from **mode**. If SHM_PERSIST is specified in **mode**, then a persistent shared memory special file is created; otherwise a non-persistent shared memory special file is created. The protection part of the **mode** parameter is modified by the process' file creation mask (see umask). When bits in **mode** other than the protection part and the SHM_PERSIST are specified in the **mode** parameter, incorrect permissions may result for the created file.

In the case of a persistent shared memory special file, the state of the file, including all data associated with the file, will persist until the file is removed or until system reboot.

The name of the shared memory special file created by the mkshm function will remain a "valid" pathname until the pathname is removed or until system reboot. After a system reboot, the pathname and file state are invalid. The file will need to be created using mkshm again. The existence of an invalid pathname is ignored by mkshm when re-creating the file by the same name.

The shared memory's owner ID and group ID will be set to the process' effective user ID, and effective group ID, respectively.

Upon successful completion, the `mkshm` function will update the `st_atime`, `st_ctime`, and `st_mtime` fields of the file (see `stat`). Also the `st_ctime` and `st_mtime` fields of the directory that contains the new entry will be updated.

DIAGNOSTICS

`mkshm` returns 0 when successful and -1 when unsuccessful. If the call fails, **errno** will contain one of the following:

EACCES	A component of the path prefix denies search permission.
EEXIST	The named file already exists.
EFBIG	The **size** argument specifies a size which exceeds the maximum shared memory special file size.
EFTYPE	The **path** refers to a file system that does not support shared memory special files.
ENAMETOOLONG	The length of the **path** string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.
ENOENT	A component of the **path** prefix does not exist or the **path** argument points to an empty string.
ENOSPC	The directory that would contain the new file cannot be extended for the file system is out of file allocation resources.
ENOTDIR	A component of the **path** prefix is not a directory.
EROFS	The directory in which the shared memory special file is to be created resides in a read-only file system.
ENOMEM	Out of a system resource.
EINVAL	Illegal **mode**, **size** less than or equal to zero.
EFAULT	The **path** argument does not point to allocated memory in the process' address space.

CONFORMANCE

`mkshm` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `chmod()`, `stat()`, `umask()`, `open()`, `close()`

The following calls operate on POSIX shared memory special files:

System Calls - `mkshm()`, `shmmap()`, `shmunmap()`, `open()`, `close()`, `unlink()`

Example - `mkshm.c`

NAME

`mmap` – map process addresses to a memory object

SYNOPSIS

```
#include <sys/mman.h>

void *mmap(addr, len, prot, flags, fildes, off)
void *addr;
size_t len;
int prot;
int flags;
int fildes;
off_t off;
```

DESCRIPTION

`mmap` establishes a mapping between the process' address space and a memory object. The `mmap` function is used as follows:

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

The mapping is established between the process' address space at an address `pa` for **`len`** bytes to the memory object represented by the **`fildes`** file descriptor at an offset **`off`** for **`len`** bytes. `pa` is the return value from the `mmap` function. The **`prot`** parameter determines the access permissions for the data being mapped. Read, write, execute, or some combination of accesses are allowed. The various flags for the protection argument are defined in `<sys/mman.h>` as follows:

Constant	Description
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

The **prot** argument can either be PROT_NONE or the bitwise inclusive OR of one or more of the other three flags. The **flags** argument specifies other information about the handling of the mapped data. **flags** can be the bitwise inclusive OR of the following flags defined in <sys/mman.h>.

Constant	Description
MAP_SHARED	Changes to the memory object are shared.
MAP_FIXED	**addr** is to be interpreted exactly.

If MAP_SHARED is specified, all write changes to the mapped data change the underlying object.

MAP_FIXED is used to request that the value of pa should be **addr** exactly. If MAP_FIXED is not used, the value of pa is chosen by the system based on the value of **addr**. If the **addr** argument is zero, the system is given a complete freedom in arriving at pa. If the **addr** argument is non-zero, its value is taken as a suggestion of a process address near which the mapping should be placed.

NOTE

LynxOS currently supports mmap only for POSIX.1b shared memory files. At this time, only the entire object can be mapped.

MAP_PRIVATE is not yet supported. Similarly, we do not support sync of only part of a mapped object or changing mapping permissions for parts of objects. PROT_NONE is not supported.

off values other than 0 are not supported.

len values other than the size of the shared memory segment are not supported.

DIAGNOSTICS

mmap establishes a mapping between the process's address space and a memory object. Upon successful completion, it returns the address at which the mapping was placed. On failure, it returns MAP_FAILED and sets **errno** as follows:

EACCES	The file descriptor **fildes** was not opened for reading. The **fildes** descriptor was not opened for writing, and PROT_WRITE was specified for a MAP_SHARED mapping.
EAGAIN	The mapping could not be locked in memory, as required by mlockall, due to a lack of resources.
EBADF	The **fildes** argument is not a valid file descriptor.
EINVAL	The **flags** argument is invalid. The arguments **addr** (if MAP_FIXED was specified) or **off** do not lie on page boundaries.
ENODEV	The **fildes** argument is a file descriptor for an invalid object.
ENOMEM	MAP_FIXED was specified, and the address range starting at **addr** and continuing for **len** bytes exceeds that allowed for the address space of a process. MAP_FIXED was not specified and there is not sufficient room in the address space for the mapping. The mapping could not be locked in memory as required by mlockall, due to a lack of space.
ENOTSUP	MAP_PRIVATE was specified in the **flags** argument, and the implementation does not support this functionality. Accesses requested in the **prot** argument are not supported.
ENXIO	The addresses in the range starting at **off** and continuing for **len** bytes are invalid for the object specified by the **fildes** descriptor. MAP_FIXED was specified in **flags**, and the combination of **addr**, **len**, and **off** are invalid for the object specified by the **fildes** descriptor.

SEE ALSO

System Calls - munmap(), ftruncate(), shm_open(), mlockall(), shm_unlink()

NAME

`mount, umount` – mount or unmount a file system

SYNOPSIS

```
mount(path, dir, readonly)
char *path, *dir;
int readonly;

umount(path)
char *path;
```

DESCRIPTION

`mount` mounts a file system at the directory named **dir**. The file named path is the special file for the block device on which the file system resides. mount interrogates path and determines if the file system is a Lynx FLFS or ISO9660 file system.

If **readonly** is non-zero, the mounted file system cannot be written to by any process.

The directory on which the file system was mounted becomes the root directory for the mounted file system. Files on the mounted system can be accessed through this directory.

The system call `umount` unmounts the file system contained on the block device **path**, or mounted on the directory named **path** (i.e. **path** may refer to the block device or to the directory). The directory on which the file system was mounted returns to its pre-mount state.

`mount` and `umount` may only be called by processes with effective user IDs of 0 (superuser processes).

DIAGNOSTICS

`mount` (`umount`) returns 0 if the file system was mounted (unmounted), or -1 if not.

If `mount` fails, **errno** will contain one of the following:

EPERM	The effective user ID of the calling process is not 0.
ENOENT	**path** does not exist.
ENOTBLK	**path** is not a block device special file.
ENOTDIR	An element of the **path** prefix is not a directory, or **dir** is not a directory.
ENAMETOOLONG	**path** or **dir** is too long.
ENOENT	**dir** does not exist.
ELOOP	Too many symbolic links were encountered when traversing either **path** or **dir**.
EFAULT	**path** or **dir** does not point to allocated memory in the process' address space.
EIO	An I/O error occurred.
ENXIO	The device does not exist.
EEXIST	The named file already exists but is not a special file.
ENOSPC	There are no more mount slots.
EBUSY	The file system is already mounted.
EBUSY	The file system has a bad super block.

If umount fails, **errno** will contain one of the following:

EPERM	The effective user ID of the calling process is not 0.
EBUSY	A process is using a file on the mounted file system.
ENOENT	**path** does not exist.
ENOTBLK	**path** is not a directory nor a block device special file.
ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.

ELOOP	Too many symbolic links were encountered when traversing the **path**.
ENXIO	The device does not exist.

NAME

`mqgetattr` – get message queue attributes (POSIX.4, Draft 9)

SYNOPSIS

```
#include <sys/mqueue.h>

int mqgetattr(fd, mqstat)
int fd;
struct mqstatus *mqstat;
```

DESCRIPTION

`mqgetattr` is used get the message queue attributes. Upon return, the following fields in the **mqstatus** structure have been set:

```
struct mqstatus {
  long    mqmaxmsg;
  long    mqrsvmsg;
  long    mqmaxbytes;
  long    mqrsvbytes;
  long    mqwrap;
  long    mqmaxarcv;
  long    mqcurmsgs;
  long    mqsendwait;
  long    mqrcvwait;
};
```

`mqmaxmsg` The maximum number of messages allowed on the queue specified by **fd**. This includes messages sent with `mqsend` or `mqputevt`. This number is the sum of the number of free messages and messages currently in use.

`mqrsvmsg` The number of messages available for use (currently free) for the queue specified by **fd**.

`mqmaxbytes` The maximum number of bytes of message data which are allowed to be send with `MSG_COPY` for queue specified by **fd**. This number may be used in a single message in the queue or in multiple smaller messages. This number is the sum of the number of free bytes and bytes currently in use.

`mqrsvbytes`	The number of bytes of message data available for use (currently free) for the queue specified by **`fd`**. Only a `mqsend()` with the `MSG_COPY` flag set consumes these bytes.
`mqwrap`	Specifies what action is to be taken when the send operation would result in the `mqrsvmsg` or `mqrsvbytes` attributes being exceeded. A value of `MQWRAP` indicates to overwrite the oldest message in the message queue. A value of `MQNOWRAP` indicates that such a send will fail. (See `mqsend`.)
`mqmaxarcv`	Specifies the maximum number of pending asynchronous `mqreceive` calls or `mqgetevt` calls allowed for this message queue.
`mqcurmsgs`	The number of messages currently on the queue.
`mqsendwait`	The number of sending processes waiting for or expecting asynchronous notification of a message delivery.
`mqrcvwait`	The number of receiving processes waiting for or expecting asynchronous notification of a message receipt.

DIAGNOSTICS

`msggetattr` returns 0 when successful and -1 when unsuccessful. If the call fails **`errno`** will contain one of the following:

`EBADF`	The **`fd`** argument is not a valid file descriptor.
`EFTYPE`	The **`fd`** argument does not reference a message queue special file.
`EFAULT`	The **`mqstat`** argument does not point to allocated memory in the process' address space.

CONFORMANCE

`mqgetattr` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `mkmq()`, `mqsetattr()`, `mqsend()`, `mqputevt()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq()`, `mqsend()`, `mqreceive()`, `mqsetattr()`, `mqgetattr()`, `mqpurge()`, `msgalloc()`, `msgfree()`, `mqputevt()`, `mqgetevt()`

Library Functions – `mqgetpid()`

Example - `mqgetattr.c`

NAME

`mqgetevt` – request asynchronous message notification (POSIX.4, Draft 9)

SYNOPSIS

```
#include <sys/mqueue.h>

int mqgetevt(fd, event)
int fd;
struct event *event;
```

DESCRIPTION

`mqgetevt` asynchronously gets an event off of the message queue. Only a message sent by `mqputevt` will satisfy this call. The actual getting of the event off of the message queue is performed by a separate thread of execution, in parallel with computation by the calling thread. The `mqgetevt` call will thus return to the calling thread before completion of the actual operation of getting the event off of the message queue; the calling thread continues with the next instruction.

`fd` specifies which message queue to get the event from. The event must be sent using a `mqputevt` call. Messages sent using `mqsend` will not satisfy the `mqgetevt` request.

When the message is available to satisfy the `mqgetevt`, an event will be sent to the calling thread.

The events structure, defined in `<sys/events.h>` contains the following fields (not necessarily in this order):

```
evt_class_t  evt_class;
void         *evt_value;
void         (*evt_handler)() ;  /* not used by LynxOS */
evtset_t     evt_classmask ;     /* not used by LynxOS */
```

Looking ahead to POSIX Draft 10, Lynx only uses the `evt_value` and `evt_class` fields of the event structure. For the same reason, we use the words "event" and "signal" interchangeably in this document, and use the `sigaction` system call to set up the action to be taken when the event is received by the calling process. The following is a sample event handler:

```
void
mqgetevt_handler(signo, value)
int  signo;
void *value;
{
  printf("in mqgetevt_handler:  ");
  printf("signo = %d, value = %d\n", signo, value);
}
```

The request shall be satisfied by the first message on the queue that was sent by `mqputevt`.

The event data is transmitted to the calling thread either as the `value` passed to the event handler or as the `value` argument of an `evtpoll`. Even if there is a waiting event on the queue at the time of the `mqgetevt` call, the value will always be returned by event notification.

The `event_value` argument of the `mqgetevt` call is not used.

Note: this function is no longer available in POSIX Draft 10.

DIAGNOSTICS

`mqgetevt` returns 0 when successful and -1 when unsuccessful. If the call fails, **`errno`** will contain one of the following:

EAGAIN	The system does not have resources to handle more receive requests for the specified queue, or to generate the event notification.
EBADF	The **`fd`** argument is not a valid file descriptor open for reading.
EFTYPE	The file specified is not a message queue special file.
EFAULT	The **`event`** argument does not point to allocated memory in the process' address space.

`EINVAL`	The `event.evt_class` value is not between `EVTCLASS_MIN` and `EVTCLASS_MAX` (the legal event values).

CONFORMANCE

`mqgetattr` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `mqputevt(), mqsetattr(), evtpoll()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()`

Library Functions - `mqgetpid()`

NAME

`mqpurge` – remove messages from a message queue (POSIX.4, Draft 9)

SYNOPSIS

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

int mqpurge(fildes, pid)
int fildes;
pid_t pid;
```

DESCRIPTION

`mqpurge` is used to remove messages from a message queue special file.

The file must have been opened with write permission.

If **`pid`** is zero and the process' effective user ID is root, then all messages on the message queue specified by **`fildes`** will be removed.

If **`pid`** is greater than zero, then all messages on the message queue specified by **`fildes`** that were sent by the process designated by **`pid`** will be removed from the message queue. Either the process ID of the calling process must equal the **`pid`** argument, or the process' effective user id must be root.

If the calling process does not have appropriate privilege, the `mqpurge` function will fail.

Note: This function will not purge events from the message queue special file.

DIAGNOSTICS

`mqpurge` returns 0 when successful and -1 when unsuccessful. If the call fails, **`errno`** will contain one of the following:

`EBADF`	The **`fildes`** argument is not a valid file descriptor open for writing.

EFTYPE	The **fildes** argument is not a message queue special file.
EPERM	The process does not have permission to purge the requested messages from the queue.
EINVAL	The **pid** argument is less than zero.

CONFORMANCE

`mqpurge` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `mqsend(), mqputevt()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()`

Library Functions - `mqgetpid()`

Example - `mqpurge.c`

NAME

`mqputevt` – put event data on a message queue (POSIX.4, Draft 9)

SYNOPSIS

```
#include <sys/mqueue.h>

int mqputevt(fd, data)
int fd;
void *data;
```

DESCRIPTION

`mqputevt` is used to add a message to the message queue that transfers only the pointer's worth of data specified by **data** as an `event_value` received during asynchronous notification. The receiving process must use the `mqgetevt` system call to establish the event notification.

The message put on the queue is equivalent to one with a message control block containing `msg_flags` set to `MSG_NOWAIT`, `msg_length` and `msg_bufsize` set to zero, and `msg_data` set to the data parameter specified in the `mqputevt` call.

None of the other fields of the message control block are relevant, since the message can only be received by a `mqgetevt` call, and only the **data** value is actually transferred to the receiver.

The `msgcb` structure is described in detail in the `mqsend` man page.

Note: this function is no longer available in POSIX Draft 10.

DIAGNOSTICS

`mqputevt` returns 0 when successful and -1 when unsuccessful. If the call fails, **errno** will contain one of the following:

`EAGAIN`	The message queue depth would be exceeded by this message.
`EBADF`	The **fd** argument is not a valid file descriptor open for writing.

`EFTYPE` The file specified is not a message queue special file.

SEE ALSO

System Calls - `mqgetevt()`, `mqsetattr()`, `mqsend()`, `mqreceive()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq()`, `mqsend()`, `mqreceive()`, `mqsetattr()`, `mqgetattr()`, `mqpurge()`, `msgalloc()`, `msgfree()`, `mqputevt()`, `mqgetevt()`

Library Functions - `mqgetpid()`

NAME

`mqreceive` – receive a message from a message queue (POSIX.4, Draft 9)

SYNOPSIS

```
#include <mqueue.h>

int mqreceive(fd, msgcbp)
int fd;
struct msgcb *msgcbp;
```

DESCRIPTION

`mqreceive` is used to synchronously or asynchronously receive a message from the message queue special file specified by **fd**. `mqreceive` will remove the message from the queue and transfer its information to the receiving process.

See the `mqsend` manual page for details of the **msgcb** structure.

The options which can be specified in the **msgcb** structure allow the user to control:

- selective receipt of messages
- whether the receiver or the system will supply the buffer to hold the message data

In the **msgcb**, only the `msg_type`, `msg_flags`, `msg_bufsize`, and `msg_data` fields have to be set up by the user for `mqreceive`. The user may also set up an optional `msg_event` field.

The `msg_type` field in the **msgcb** structure is used to selectively receive messages from the message queue. `msg_type` specifies the type of the message to be received:

- If `msg_type` is equal to zero, the oldest message on the queue is received.
- If `msg_type` is less than zero, the oldest message of the lowest valued type currently on the queue that is less than or equal to the absolute value of `msg_type` is received.

- If `msg_type` is greater than zero, the first message of type equal in value to `msg_type` is received.

The `msg_data` and `msg_bufsize` fields are used to control the buffer into which the message is to be received.

- If `msg_data` is set to NULL, the receiver is requesting the message buffer to be supplied by the system. The receiver should use `msgfree` to free this buffer when it is no longer needed.
- If `msg_data` is non-NULL, and the `msg_bufsize` field is non-NULL, the `msg_bufsize` field defines a buffer within the user data space into which the message data will be copied on receipt.
- If `msg_data` is non-NULL, and the `msg_bufsize` field is NULL, the sender's `msg_data` value is copied into the receiver's `msg_data` field and the `MSG_OVERRIDE` flag will be set in the `msg_flags` field of the receiver's **`msgcb`**. Note that if the sender is only sending a pointer's worth of data and the receiver's `msg_data` is set to NULL, then the system will supply a buffer, the data will be copied to the supplied buffer and the `MSG_OVERRIDE` flag will NOT be set. Simply setting the `msg_data` field to = (void *)1 will prevent the system from supplying the unneeded buffer.
- If the receiver specifies a buffer and `MSG_MOVE` was specified on the send of the message being received, then the system will free the sender's buffer after the message data has been transferred to the receiver.

The `msg_flags` field in the **`msgcb`** controls options on the receive. The `msg_flags` argument is constructed from the bitwise inclusive OR of zero or more of the following flags:

`MSG_ASYNC` If `MSG_ASYNC` is set, the `mqreceive` returns without a message being received, but an asynchronous receive will be pending. When a message of the required type exists on the queue, the pending receive will be satisfied and asynchronous notification will occur using the values set by the calling process in the `msg_event` structure of the **`msgcb`**.

Note the directions below for setting up the event structure and the actions needed to ensure correct event notification. If `MSG_ASYNC` is not set and `O_NONBLOCK` is not set, the `mqreceive` will not return until a message of the requested type exists on the queue to satisfy the receive. If `MSG_ASYNC` is not set and `O_NONBLOCK` is set, the `mqreceive` returns without waiting for a message of the requested type, and will be successful if a message was received, and unsuccessful if no message was received.

`MSG_TRUNC` If `MSG_TRUNC` is set, a message that is too large for the receiver's buffer is truncated to fit within the `msg_bufsize` bytes specified in **`msgcb`**. When `MSG_TRUNC` is not set, a message that is larger than the buffer, but otherwise would satisfy the `mqrecieve` causes an error to be returned; the `mqreceive` fails with **`errno`** of `E2BIG`, the message remains in the message queue, but the `msg_length` is set to the size of the message.

Upon return or after asynchronous notification, the following fields within the **`msgcb`** have been set:

`msg_bufsize` For system allocated buffers, `msg_bufsize` is set to the maximum size message that can be contained in the buffer. This will be set to zero if only a pointer's worth of data was received in the `msg_data` field.

`msg_length` If `msg_bufsize` is non-zero, and the `mqreceive` succeeded, `msg_length` is set to the actual size of the message received. If `msg_bufsize` is non-zero, and the `mqreceive` failed with **`errno`** `E2BIG`, `msg_length` is set to the minimum size the buffer needed to be in order to prevent the `E2BIG` error. If the sender's `msg_bufsize` was zero, the receiver's `msg_length` field is set to the sender's `msg_length` field value.

`msg_type` `msg_type` is set to the type of the message just received.

`msg_timesent` `msg_timesent` is set to indicate the time that the message was sent.

`msg_sender`	`msg_sender` is set to a unique sender identifier.
`msg_errno`	`msg_errno` is set to indicate errors that occurred during asynchronous receipt.
`msg_data`	Set to point to the message data, or contains a pointer's worth of data.
`msg_flags`	If `msg_data` had been set in the receive to non-NULL and the sender had sent a pointer's worth of data, `MSG_OVERRIDE` is set to indicate that the receiver did not use the buffer referenced by `msg_data` and in fact has overwritten the contents of the `msg_data` field (the pointer provided by the receiver).

The `events` structure, defined in `<sys/events.h>` contains the following fields (not necessarily in this order):

```
evt_class_t   evt_class;
void         *evt_value;
void         (*evt_handler)() ;  /* not used by LynxOS */
evtset_t      evt_classmask ;       /* not used by LynxOS */
```

Looking ahead to POSIX Draft 10, Lynx only uses the `evt_value` and `evt_class` fields of the event structure. For the same reason, we use the words “event” and “signal” interchangeably in this document, and use the `sigaction` system call to set up the action to be taken when the event is received by the calling process. The following is a sample event handler:

```
void
async_mqreceive_handler(signo, value)
int  signo;
void *value;
{
  printf("in async_mqreceive_handler:  ");
  printf("signo = %d, value = %d\n", signo, value);
}
```

DIAGNOSTICS

Upon successful completion of a either a synchronous or an asynchronous receive, a value of 0 is returned. Otherwise a value of -1 is returned and **errno** is set to indicate the error and no message is removed from the queue.

For an asynchronous operation, the `mqreceive` may fail following the successful return from the call. Upon asynchronous notification indicating a receive termination, `msg_errno` is set to zero to indicate successful receipt of a message or is set to indicate the error if no message was received. If an error is indicated, no message is removed from the queue.

The **errno** or `msg_errno` fields in the **msgcb** may contain one of the following:

E2BIG	A message was available, `MSG_TRUNC` was reset in the call, the `msg_data` field is non-NULL, and `msg_bufsize` defines a buffer that was too small to receive the message.
EAGAIN	`MSG_ASYNC` was not set and `O_NONBLOCK` was set, and no message of the requested type exists on the queue. `EAGAIN` is also set if `MSG_ASYNC` is set but there are no resources available to handle the asynchronous request.
EBADF	The **fd** argument is not a valid file descriptor open for reading.
EFTYPE	The **fd** argument is not a message queue special file.
EINTR	An event or signal interrupted this `mqreceive`.
ENOMEM	A message was available, the `msg_data` field in **msgcb** was NULL, and the implementation could not supply the buffer.
EFAULT	The **msgcbp** argument does not point to allocated memory in the process' address space. The receiver has accidentally provided the sender's buffer.
EMFILE	The system was unable to map the allocated buffer to the receiving process' address space (this may occur if the sender set the `msg_flags` to `MSG_MOVE` or if the receiver has set `msg_data` to NULL). In the case of the receiver setting `msg_data` to NULL, `EMFILE` may be returned if the system is out of memory.

CONFORMANCE

`mqreceive` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`msgfree(), mqsend()`

Calls that operate on POSIX message queue special files include:

System Calls -	`mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()`
Library Functions -	`mqgetpid()`
Example -	`mqreceive.c`

NAME

mqsend – send a message to a message queue (POSIX.4, Draft 9)

SYNOPSIS

```
#include <mqueue.h>

int mqsend(fd, msgcbp)
int fd;
struct msgcb *msgcbp;
```

DESCRIPTION

mqsend delivers a message to the message queue special file specified by **fd**. The message and options for the mqsend are specified within the **msgcb** structure referred to by **msgcb**.

The message control block (**msgcb**) structure contains the following fields, not necessarily in this order:

```
struct event    msg_event;
int             msg_errno;
int             msg_retval;
struct ar       *khook;  /* not used for POSIX messages */
unsigned int    msg_flags;
long            msg_type;
size_t          msg_length;
size_t          msg_bufsize;
void            *msg_data;
struct timespec msg_timesent;
sender_t         msg_sender;
```

The options which can be specified in the **msgcb** structure allow the user to control:

- how the message data is stored.
- when the message buffer will become available again.
- how many copies of the buffer will need to be done by the system.
- whether the sender will wait for the message to be received.

- whether the sender expects to be informed that the message has been received.
- what action should be taken when there are no resources available to send the message.
- a message tag, `msg_type`, used to help the receiver handle selective message receipt.

The **msgcb** structure contains the following fields (note that not all fields are used for `mqsend`; they are included here for reference):

`msg_flags`	Contains the operation control flags (see below).
`msg_type`	A sender-supplied value which may be used for selective receipt of the message.
`msg_length`	The number of bytes of valid message data.
`msg_bufsize`	The size of the data buffer (or zero if `msg_data` contains a pointer's worth of data).
`msg_data`	The pointer to the buffer holding the data (or the data itself if `msg_data` contains a pointer's worth of data).
`msg_event`	An event structure to be used for asynchronous notification.
`msg_errno`	Provides error information with asynchronous receive.
`msg_timesent`	The time at which the message was sent.
`msg_sender`	A unique system-wide identification of the sending process.

Only the `msg_flags`, `msg_type`, `msg_length`, `msg_bufsize`, and `msg_data` fields have to be set up by the user for `mqsend`. The user may also set up an optional `msg_event` field.

All other fields in the **msgcb** structure will be ignored by `mqsend` and the **msgcb** structure will be unchanged on return from `mqsend` and available for reuse.

The `msg_data` field can either contain a pointer to the data associated with the message or it can contain a pointer's worth of data.

To specify that the msg_data field contains a pointer to message data:

- Set msg_bufsize to the size of the data buffer.
- Set msg_length to the size in bytes of the data to be sent.
- Set msg_data to contain a pointer to the data to be sent (a pointer to the data buffer).

To pass a pointer's worth of data in the msg_data field:

- Set msg_bufsize to zero.
- Set the msg_data to contain the data instead of a pointer to the data.

The semantics of the pointer's worth of data are determined by the sender and receiver of the message.

Note that a possible use of passing a pointer's worth of data is in using shared memory to pass the data. If the sending and receiving processes both have access to the same shared memory area, then the sending process can put the address of the data in msg_data and the length of the data in msg_length. In this case, the sender must not modify the data until the receiver is finished with it. (If the receiver specifies a buffer, the MQ_OVERRIDE flag can be checked to recognize that the message being received is in shared memory instead of the specified buffer (see mqreceive)).

The msg_type field in the **msgcb** structure is used for selective receipt of a message. It is passed as set by the caller of mqsend to the receiver of the message. The msg_type value specified in mqsend must be a positive integer.

The msg_flags are operation control flags. There are two categories of msg_flags:

1. "wait" flags which specify whether the sender waits for the message to be received and whether the sender expects notification of message receipt.

2. “copy” flags which specify how the sender’s data buffer is to be used. The major issues in this area are how many times the data must be copied by the system and when/if the sender’s buffer can be re-used.

The `msg_flags` argument is constructed from the bitwise inclusive OR of one or more of the “wait” and “copy” flags.

Exactly one of the following three “wait” flags must be specified:

`MSG_WAIT` When `MSG_WAIT` is set, the `mqsend` does not return until the message has been received.

`MSG_ASYNC` When `MSG_ASYNC` is set, the `mqsend` returns without waiting for the message to be received, but an asynchronous event notification will occur when the message has been received. The `msg_event` structure values are supplied by the sender. Note the directions below for setting up the event structure and the actions needed to ensure correct event notification.

`MSG_NOWAIT` When `MSG_NOWAIT` is set, the `mqsend` returns without waiting for the message to be received, and no event notification is generated when the message is received.

Exactly one of the following three “copy” flags must be specified IF `msg_bufsize` is non-zero (`msg_data` contains a pointer to data):

`MSG_MOVE` When `MSG_MOVE` is set, and `msg_bufsize` is non-zero, the buffer referenced by the **`msgcb`** structure may no longer be accessible by the sender upon return from the `mqsend`. The buffer must have been allocated with `msgalloc`. The sender should not attempt to access the buffer following the send, even to free it. The receiver is responsible for freeing the buffer (or the system will free the buffer if the receiver has supplied its own buffer).

This message-sending technique allows the buffered data to be passed to the receiver without any copies being made.

MSG_COPY — When MSG_COPY is set, and msg_bufsize is non-zero, the buffer is available for re-use upon return from mqsend. In this case, the data is copied twice: once to kernel space and a second time to the receiver's buffer.

MSG_USE — When MSG_USE is set, and msg_bufsize is non-zero, the buffer will be accessed during the mqreceive of the corresponding message. Therefore, an application using an asynchronous send should not alter addressability to the buffer or change the buffer contents before the message has been received. Receipt can be determined using asynchronous notification, or some other method agreed upon by the cooperating sending and receiving processes. In this case, a single copy is made from the send buffer to the receive buffer when the receive occurs.

Note that messages sent by a process with MSG_USE will be removed from the message queue when the process closes the message queue special file or when the process exits (see close).

If the message being sent would cause either the mqmaxbytes or mqmaxmsg limits on the queue to be exceeded, and the queue wrap attribute is set, then as many of the oldest messages as necessary will be removed to provide enough resources to queue the new message.

If the message being sent would cause any of the above limits to be exceeded, and the queue wrap attribute is not set, then O_NONBLOCK being set causes the mqsend to return an error without blocking, but O_NONBLOCK not being set causes the mqsend to block until the message can be queued without exceeding the limits.

The events structure, defined in <sys/events.h> contains the following fields (not necessarily in this order):

```
evt_class_t  evt_class;
void        *evt_value;
void        (*evt_handler)() ; /* not used by LynxOS */
evtset_t    evt_classmask ;    /* not used by LynxOS */
```

Looking ahead to POSIX Draft 10, Lynx only uses the evt_value and evt_class fields of the event structure. For the same reason, we use the words “event” and “signal” interchangeably in this document, and use the sigaction system call to set up the action to be taken when the event is received by the calling process.

The following is a sample event handler:

```
void
async_mqsend_handler(signo, value)
int  signo;
void *value;
{
  printf("in async_mqsend_handler:  ");
  printf("signo = %d, value = %d\n", signo, value);
}
```

DIAGNOSTICS

mqsend returns 0 when successful and -1 when unsuccessful. If the call fails, **errno** will contain one of the following:

EAGAIN	O_NONBLOCK is set on the **fd** file descriptor, and the queue wrap attribute is not set for the corresponding message queue, and at least one of the maximum limits of the queue would be exceeded by the message being sent. EAGAIN is also set if MSG_ASYNC is set but there are no resources available to handle the asynchronous request.
EBADF	The **fd** argument is not a valid file descriptor open for writing.
EFTYPE	The **fd** argument is not a message queue special file.
EINTR	An event or signal interrupted this mqsend.
EFAULT	The **msgcbp** argument does not point to allocated memory in the process’ address space.
EINVAL	MSG_MOVE is set in the msg_flags field of **msgcb**, msg_bufsize is not zero (we are not just passing a pointer’s worth of data), and the message buffer was not allocated by msgalloc. The msg_type argument is not

a positive integer. More than one of `MSG_WAIT`, `MSG_ASYNC`, and `MSG_NOWAIT` is set. More than one of `MSG_MOVE`, `MSG_COPY`, and `MSG_USE` is set. `msg_data` is a pointer and `msg_length` is less than or equal to zero.

CONFORMANCE

`mqsend` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `msgalloc(), mqreceive(), mqsetattr()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()`

Library Functions - `mqgetpid()`

Example - `mqsend.c`

NAME

`mqsetattr` – set message queue attributes (POSIX.4, Draft 9)

SYNOPSIS

```
#include <sys/mqueue.h>

int mqsetattr(fd, mqstat)
int fd;
struct mqstatus *mqstat;
```

DESCRIPTION

`mqsetattr` is used to modify attributes associated with the message queue specified by **fd**. The message queue attributes corresponding to the following fields defined in the **mqstatus** structure are set to the specified values.

See the `mqgetattr` manual page for details of the **mqstatus** structure.

`mqmaxmsg`	The maximum number of messages allowed on the queue specified by **fd**. This includes messages sent with `mqsend` or `mqputevt`. This number is the sum of the number of free messages and messages currently in use.
`mqrsvmsg`	The number of messages available for use (currently free) for the queue specified by **fd**.
`mqmaxbytes`	The maximum number of bytes of message data which are allowed to be sent with `MSG_COPY` for queue specified by **fd**. This number may be used in a single message in the queue or in multiple smaller messages. This number is the sum of the number of free bytes and bytes currently in use.
`mqrsvbytes`	The number of bytes of message data available for use (currently free) for the queue specified by **fd**. Only a `mqsend()` with the `MSG_COPY` flag set consumes these bytes.

`mqwrap` Specifies what action is to be taken when the send operation would result in the `mqrsvmsg` or `mqrsvbytes` attributes being exceeded. A value of `MQWRAP` indicates to overwrite the oldest message in the message queue. A value of `MQNOWRAP` indicates that such a send will fail. (See `mqsend`.)

`mqmaxarcv` Specifies the maximum number of pending asynchronous `mqreceive` calls or `mqgetevt` calls allowed for this message queue.

Note that the values of `mqrsvmsg` and `mqmaxmsg` must be changed in tandem. For instance, if `mqrsvmsg` = 5, and `mqmaxmsg` = 10, to change the increase the number of messages by 5, increase both the values of `mqrsvmsg` and `mqmaxmsg` by 5. The same is true for `mqrsvbytes` and `mqmaxbytes`.

All other fields within the **`mqstatus`** structure are ignored by `mqsetattr`. The other fields in **`mqstatus`** structure are described in the `mqgetattr` man page.

To modify the attributes of the message queue special file, one must either be the owner of the file (same effective user ID as the creator of the file), or must have an effective user ID of root.

Note: the `mqmaxbytes` and `mqrsvbytes` fields are usually incremented or decremented in whole packets of bytes, not individual bytes. Thus, one may get an actual increment much larger than the requested increment; or one may request a small decrement and see no change in these fields where the decrement was small enough to not specify a whole packet.

DIAGNOSTICS

`msgsetattr` returns 0 when successful and -1 when unsuccessful. If the call fails, **`errno`** will contain one of the following:

`EBADF` The **`fd`** argument is not a valid file descriptor.

`EFTYPE` The **`fd`** argument does not reference a message queue special file.

EINVAL	The value of one of the queue attributes is too small to support the current contents of the message queue. For instance, requesting a `mqmaxmsg` lower than the number of messages currently on the queue will cause `mqsetattr` to fail.
ENOMEM	The value of one of the queue attributes would cause some system address limit or resource limit to be exceeded.
EPERM	The calling process does not have appropriate privileges to alter the attributes of the message queue specified by **`fd`**.
EFAULT	The **`mqstat`** argument does not point to allocated memory in the process' address space.

NOTES

If `mqsetattr()` fails, you may want to re-set the queue's attributes to the prior value; part of `mqsetattr` can fail after another part succeeds.

The number of bytes of message data allocated for the queue may not match the number requested.

CONFORMANCE

`mqsetattr` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `mkmq(), mqsetattr(), mqsend(), mqputevt()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(),` `msgalloc()`, `msgfree()`, `mqputevt()`, `mqgetevt()`

Library Functions – `mqgetpid()`

Example - `mqsetattr.c`

NAME

`msgalloc` – allocate an optional message buffer (POSIX.4, Draft 9)

SYNOPSIS

```
#include <sys/mqueue.h>

int msgalloc(msgcbp)
struct msgcb *msgcbp;
```

DESCRIPTION

`msgalloc` is used to allocate a buffer for the optional message data part of a message.

The **msgcb** structure is described in detail in the `mqsend` man page.

The `msg_length` field in the **msgcb** structure contains the number of bytes requested.

Upon return from `msgalloc`, the `msg_bufsize` field in **msgcb** is set to the allocated size which is at least `msg_length` bytes long and the `msg_data` field contains a pointer to the allocated buffer. The system will allocate a minimum of one page of memory per buffer: 4096 bytes.

NOTE

When sending a message (`mqsend`) where the data buffer has been allocated using `msgalloc`, the `MSG_MOVE` flag must be specified. The sender should not attempt to access the buffer following the send, even to free it.

If the receiver has supplied a buffer for the data to be copied into, then once the data has been transferred to the receiver, the system will free the sender's buffer. If the receiver has requested that the message buffer be supplied to it, then the receiver must call `msgfree` when done with the buffer to free it.

DIAGNOSTICS

`msgalloc` returns 0 when successful and -1 when unsuccessful. If the call fails, **`errno`** will contain one of the following:

`ENOMEM`	Insufficient memory to allocate the buffer.
`EINVAL`	The `msg_length` is zero.
`EFAULT`	The **`msgcb`** argument does not point to allocated memory in the process' address space.
`EMFILE`	The system is out of memory.

SEE ALSO

System Calls - `msgfree(), mqsend(), mqreceive()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()`

Library Functions - `mqgetpid()`

NAME

msgctl – message control (System V-compatible)

SYNOPSIS

```
#include <ipc.h>
#include <msg.h>

int msgctl(msqid, cmd, buff)
int msqid;
int cmd;
struct msqid_ds *buff;
```

DESCRIPTION

msgctl provides various controls over a System V-compatible message queue. The argument **msqid** specifies the message queue to be manipulated. The operation is given in **cmd** as follows:

IPC_STAT — Populate the fields of structure pointed to by **buff** with the data in the msqid_ds structure associated with **msqid**.

IPC_SET — Populate the following fields of the msqid_ds structure associated with **msqid** with the corresponding fields of the structure pointed to by **buff**:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode (low-order 9 bits)
msg_qbytes
```

IPC_RMID — Remove the message queue specified by **msqid**.

The IPC_SET and IPC_RMID commands can only be executed by a process with an effective user ID equal either to super-user or to the value of msg_perm.cuid or msg_perm.uid in the msqid_ds structure associated with **msqid**. Moreover, in the IPC_SET command, only super-user can increase the value of msg_qbytes.

The `msqid_ds` structure is defined in `msg.h` as follows:

```
struct msqid_ds {
  struct ipc_perm msg_perm;   /* permissions */
  char *msg_first;   /* first message in queue */
  char *msg_last;     /* last message in queue */
  unsigned short msg_cbytes;   /* number bytes on queue */
  unsigned short msg_qnum;  /* # of messages on queue */
  unsigned short msg_qbytes;  /* max # of bytes on queue */
  unsigned short msg_lspid;  /* last sender */
  unsigned short msg_lrpid;  /* last receiver */
  unsigned long msg_stime;   /* last send time */
  unsigned long msg_rtime;   /* last receive time */
  unsigned long msg_ctime;  /* creation time */
};
```

DIAGNOSTICS

`msgctl` returns 0 if successful, or -1 if not. If `msgctl` fails, **`errno`** will contain one of the following:

`EINVAL`	**`msqid`** is not a valid message queue identifier.
`EINVAL`	**`cmd`** is not a valid command.
`EACCES`	**`cmd`** is equal to `IPC_STAT` and read permission is denied to the calling process.
`EPERM`	**`cmd`** is equal to `IPC_RMID` or `IPC_SET` and the effective user ID of the calling process is neither super-user nor equal to `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` structure associated with **`msqid`**.
`EPERM`	A process is trying to increase the value of `msg_qbytes` using the `IPC_SET` command, but its effective user ID is not super-user.
`EFAULT`	**`buff`** points to an illegal address.

NAME

msgfree – deallocate msg_data buffer (POSIX.4, Draft 9)

SYNOPSIS

```
int msgfree(msgcbp)
struct msgcb *msgcbp;
```

DESCRIPTION

msgfree is used to deallocate the buffer referenced by the msg_data field of **msgcb**. The buffer must have either been allocated by msgalloc, or by the system during a mqreceive call that had a NULL msg_data field (see mqreceive).

The **msgcb** structure is described in detail in the mqsend man page.

DIAGNOSTICS

msgfree returns 0 when successful and -1 when unsuccessful. If the call fails, **errno** will contain one of the following:

EINVAL	The buffer specified in msg_data was not a buffer allocated by either msgalloc or by the system for a mqreceive.
EFAULT	The **msgcb** argument does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - msgalloc(), mqsend(), mqreceive()

Calls that operate on POSIX message queue special files include:

System Calls - mkmq(), mqsend(), mqreceive(), mqsetattr(), mqgetattr(), mqpurge(), msgalloc(), msgfree(), mqputevt(), mqgetevt()

Library Functions - mqgetpid()

NAME

`msgget` – get message queue (System V-compatible)

SYNOPSIS

```
#include <ipc.h>
#include <msg.h>

int msgget(key, flag)
int key;
int flag;
```

DESCRIPTION

`msgget` returns the message queue identifier associated with **`key`**. If the message queue exists, its identifier is returned; if not, a new message queue is created as long as the `IPC_CREAT` bit is set in **`flag`**.

A new message queue will always be created if **`key`** is equal to `IPC_PRIVATE`.

When a new message queue is created, its associated `msqid_ds` data structure is initialized as follows:

`msg_perm.cuid`	set equal to the effective user ID.
`msg_perm.uid`	set equal to the effective user ID.
`msg_perm.cgid`	set equal to the effective group ID.
`msg_perm.gid`	set equal to the effective group ID.
`msg_perm.mode`	low-order 9 bits set equal to low-order 9 bits of **`flag`**.
`msg_qnum`	set equal to zero.
`msg_lspid`	set equal to zero.
`msg_lrpid`	set equal to zero.
`msg_stime`	set equal to zero.
`msg_rtime`	set equal to zero.

`msg_ctime`	set equal to the current time.
`msg_qbytes`	set equal to the system limit.

DIAGNOSTICS

`msgget` returns the message queue identifier if successful, or `-1` if not. If `msgget` fails, **`errno`** will contain one of the following:

`EACCES`	A message queue exists for **`key`**, but operation permission is denied according to the low-order 9 bits of **`flag`**.
`ENOENT`	No message queue exists for **`key`** and the `IPC_CREAT` bit is not set in **`flag`**.
`ENOSPC`	A message queue is to be created, but there are no more message queue identifiers.
`EEXIST`	A message queue exists for **`key`** but the `IPC_CREAT` and `IPC_EXCL` bits are both set in **`flag`**.

NAME

`msgop: msgsnd, msgrev` – message operations (System V-compatible)

SYNOPSIS

```
#include <ipc.h>
#include <msg.h>

int msgsnd(msqid, msgp, msgsz, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
int msgflg;

int msgrcv(msqid, msgp, msgsz, msgtyp, msgflg)
int msqid;
struct msgbuf *msgp;
int msgsz;
long msgtyp;
int msgflg;
```

DESCRIPTION

`msgsnd` sends a message to the queue associated with identifier **msqid**. The message resides in a structure pointed to by **msgp**, defined in `msg.h` as follows:

```
struct msgbuf {
  long mtype;     /* message type */
  char mtext[];   /* message text */
};
```

The `mtype` field is the message type and is used by the receiving process when it calls `msgrcv`. The `mtext` field is the message text and should be **msgsz** characters long.

msgflg is used only if the number of bytes on the queue has reached its limit (`msg_qbytes`) or if the total number of messages on all system message queues has reached its limit. In these cases, **msgflg** determines whether or not the process gets suspended as follows:

- If the `IPC_NOWAIT` bit is set in **`msgflg`**, the `msgsnd` call fails with a return value of -1 and **`errno`** set to `EAGAIN`.
- If the `IPC_NOWAIT` bit is not set in **`msgflg`**, the process is suspended until one of the following occurs:
- The condition responsible for the suspension no longer exists, in which case the message is sent.
- The message queue associated with **`msqid`** is removed from the system, in which case a value of -1 is returned with **`errno`** set to `EIDRM`.
- The process receives a signal that is to be caught, in which case the message is not sent and a value of 1 is returned with **`errno`** set to `EINTR`.

If `msgsnd` succeeds, the `msqid_ds` structure associated with **`msqid`** is modified as follows:

`msg_qnum`	incremented by 1.
`msg_lspid`	set equal to the process ID of the calling process.
`msg_stime`	set equal to the current time.

`msgrcv` receives a message from the queue associated with identifier **`msqid`** and copies it into the structure pointed to by **`msgp`**, defined in `msg.h` as follows:

```
struct msgbuf {
  long mtype;     /* message type */
  char mtext[];   /* message text */
};
```

The `mtype` field is the message type and is set by the sending process. The `mtext` field is the message text.

If the size of the message, as specified by the sending process, is larger than **`msgsz`**, one of the following occurs:

- If the `MSG_NOERROR` bit is set in **`msgflg`**, the message is truncated to **`msgsz`**.

- If the `MSG_NOERROR` bit is not set in **`msgflg`**, the `msgrcv` call fails with a return value of -1 and **`errno`** set to `E2BIG`.
- **`msgtyp`** specifies the type of message to receive as follows:
- If **`msgtyp`** is 0, the first message on the queue is received.
- If **`msgtyp`** is greater than 0, the first message of type **`msgtyp`** is received.
- If **`msgtyp`** is less than 0, the first message of the lowest type that is less than or equal to the absolute value of **`msgtyp`** is received.

If no message of the desired type is on the queue, then **`msgflg`** determines whether or not the process gets suspended as follows:

- If the `IPC_NOWAIT` bit is set in **`msgflg`**, the `msgrcv` call fails with a return value of -1 and **`errno`** set to `ENOMSG`.
- If the `IPC_NOWAIT` bit is not set in **`msgflg`**, the process is suspended until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue associated with **`msqid`** is removed from the system, in which case a value of -1 is returned with **`errno`** set to `EIDRM`.
  - The process receives a signal that is to be caught, in which case no message is received and a value of -1 is returned with **`errno`** set to `EINTR`.

If `msgrcv` succeeds, the `msqid_ds` structure associated with **`msqid`** is modified as follows:

`msg_qnum`	decremented by 1.
`msg_lrpid`	set equal to the process ID of the calling process.
`msg_rtime`	set equal to the current time.

DIAGNOSTICS

`msgsnd` returns 0 if successful, or -1 if not.

If `msgsnd` fails, no message is sent, and **errno** will contain one of the following:

EINVAL	**msqid** is not a valid message queue identifier.
EINVAL	`mtype` is less than 1.
EINVAL	**msgsz** is less than zero or greater than the system limit.
EACCES	Operation permission is denied to the calling process.
EAGAIN	The message cannot be sent and the `IPC_NOWAIT` bit is set in **msgflg**.
EFAULT	**msgp** points to an illegal address.
EIDRM	The calling process was suspended waiting to send and the message queue associated with msqid is removed from the system.
EINTR	The calling process was suspended waiting to send and it receives a signal that is to be caught.

`msgrcv` returns the actual number of characters placed into `mtext` if successful, or -1 if not.

If `msgrcv` fails, no message is received, and **errno** will contain one of the following:

EINVAL	**msqid** is not a valid message queue identifier.
EINVAL	**msgsz** is less than 0.
EACCES	Operation permission is denied to the calling process.
E2BIG	The size of the received message, as specified by sending process, is greater than **msgsz** and the `MSG_NOERROR` bit is not set in **msgflg**.
ENOMSG	The queue does not contain a message of the desired type and the `IPC_NOWAIT` bit is set in **msgflg**.

EFAULT	**msgp** points to an illegal address.
EIDRM	The calling process was suspended waiting to receive and the message queue associated with msqid is removed from the system.
EINTR	The calling process was suspended waiting to receive and it receives a signal that is to be caught.

NAME

munmap – unmap previously mapped addresses

SYNOPSIS

```
#include <sys/mman.h>

void *munmap(addr, len)
void *addr;
size_t len;
```

DESCRIPTION

munmap removes all mappings, which were established by previous mmap calls, for pages containing any part of the process' address space starting at **addr** and continuing for **len** bytes. Further references to these pages will result in segmentation faults, causing SIGSEGV to be delivered to the process. Also, all memory locks associated with this range are removed.

NOTE

LynxOS currently only supports mapping and unmapping of an entire shared memory object. LynxOS currently does not support private mappings.

DIAGNOSTICS

munmap removes the specified mappings when successful, and returns 0. On error it returns -1, and sets **errno** as follows:

EINVAL Some of the addresses in the specified range are outside the range allowed for the process' address space. The **addr** argument does not lie on a page boundary.

SEE ALSO

System Calls - mmap(), ftruncate(), shm_open(), mlockall()

NAME

nanosleep – high resolution sleep

SYNOPSIS

```
#include <time.h>

int nanosleep(rqtp, rmtp)
const struct timespec *rqtp;
struct timespec *rmtp;
```

DESCRIPTION

nanosleep causes the calling process to be suspended from execution for the specified time. The **rqtp** argument specifies the amount of time for which the process will be suspended. However if a signal is delivered to the calling process, which is either caught or terminates the process before the nanosleep duration expires, the process suspension duration will decrease. If the **rmtp** argument is not NULL and if nanosleep is interrupted by a signal, the location referenced by the **rmtp** argument is set to contain the amount of time remaining in the interval.

The nanosleep function does not interfere with the action or blocking of any signal.

DIAGNOSTICS

If the requested time elapses, nanosleep returns a value of 0. If the nanosleep function fails, it returns -1 and sets **errno** as follows:

EINVAL	The **rqtp** argument specified a nanosecond value less than zero or greater than or equal to 1000 million.
EINTR	nanosleep function was interrupted by a signal.

CONFORMANCE

The nanosleep function is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library calls - `sleep()`

Example - `nanosleep.c`

NAME

`newconsole` – choose a new console

SYNOPSIS

```
int newconsole(fd)
int fd;
```

DESCRIPTION

`newconsole` closes the old console device and sets up the character device specified by the open file descriptor **fd** as the new console device. The effective user ID of the calling process must be super-user to use this system call.

DIAGNOSTICS

`newconsole` returns 0 if the new console was set up, or -1 if not.

If `newconsole` fails, **errno** will contain one of the following:

EPERM	The effective user ID of the calling process is not super-user.
EBADF	**fd** is not a valid file descriptor.
EINVAL	**fd** does not refer to a character device.

SEE ALSO

System Calls - `open()`

NAME

nfsmount – mount a network file system (NFS) directory

SYNOPSIS

```
#include <rpc/rpc.h>
#include <nfs/nfs_prot.h>

nfsmount(sock_addr_ptr, file_handle_ptr, dir, readonly, hard)
struct sockaddr_in *sock_addr_ptr;
nfs_fh *file_handle_ptr;
char *dir;
int readonly;
int hard;
```

DESCRIPTION

nfsmount mounts a remote network file system (NFS) directory at the directory named **dir**. **sock_addr_ptr** specifies the network address of the NFS server and **file_handle_ptr** specifies the file handle for the remote directory on the NFS server. The file handle should be obtained by exchanging NFS Mount protocol messages with the remote NFS server.

If **readonly** is non-zero, the mounted file system cannot be written to by any process.

The parameter hard is currently unused and should be 0.

nfsmount may only be called by processes with effective user IDs of 0 (super-user processes).

DIAGNOSTICS

nfsmount returns 0 if the remote file system directory was mounted, or -1 if not.

If nfsmount fails, **errno** will contain one of the following:

EPERM	The effective user ID of the calling process is not 0.
ENOTDIR	An element of **dir** is not a directory.

ENAMETOOLONG	**dir** is too long.
ENOENT	**dir** does not exist.
ELOOP	Too many symbolic links were encountered when traversing **dir**.
EFAULT	**dir** does not point to allocated memory in the process' address space.
EIO	An I/O error occurred.
ENXIO	The device does not exist.
ENOSPC	There are no more mount slots.
EBUSY	The **dir** is not free for mounting.

NAME

open – open a file

SYNOPSIS

```
#include <file.h>
#include <sys/types.h>
#include <fcntl.h>

int open(path, oflags [mode])
char *path;
int oflags;
mode_t mode;
```

DESCRIPTION

open opens the file named in **path** for reading and/or writing, depending on the argument **oflags**. The value of **oflags** indicates the intended use of the file and may be the bitwise-OR of any number of the following. Exactly one of the flags O_RDONLY, O_WRONLY, and O_RDWR shall be specified in **oflags**:

O_APPEND — Append each write to the end of the file.

O_CREAT — If the file does not exist, create it.

O_DSYNC — When opening a regular file, this flag affects subsequent writes. If set, each write() will wait for the file data to be physically written onto storage.

O_EXCL — Return an error if file exists.

O_FSYNC — When opening a regular file, this flag affects subsequent writes. If set, each write() will wait for the file status and file data to be physically written onto storage.

O_NDELAY — Future reads and writes may be effected by this flag.

When open is called with **path** pointing to a FIFO file and the O_RDONLY and O_NDELAY flags set in **oflags** the open returns immediately. If the O_WRONLY and O_NDELAY flags are set the open will return an error if the file is not opened for reading by another process. If only `O_RDONLY` or `O_WRONLY` are set then the `open` will block and wait for another process to open the file `O_WRONLY` or `O_RDONLY`.

When `open` is called with a **path** that is associated with a communications line and the `O_NDELAY` flag is not set in **oflags** the `open` will block and wait for a carrier. If `O_NDELAY` is set the `open` will not wait.

`O_NOCTTY`	If **path** is a terminal this flag prevents that terminal from becoming the control terminal for this process.
`O_NONBLOCK`	POSIX flag equivalent to `O_NDELAY`.
`O_RDONLY`	Open for reading only.
`O_RDWR`	Open for both reading and writing.
`O_TRUNC`	If **path** is the name of a regular file truncate the file to size zero.
`O_WRONLY`	Open for writing only.

If the file is created as a result of the `open` call, the permissions for the new file are set to the value obtained by the bitwise-AND of **mode** and the complement of the current **umask**.

`open` returns an integer `file descriptor` if it succeeds. The file descriptor is used to reference the file in various other system calls. If the `O_APPEND` bit is set in **oflags**, the file position (the "seek pointer") is set to the end of the file. In all other cases the file position is set to the beginning of the file.

For POSIX shared memory special files, the following exceptions apply:

- If **path** refers to an existing, valid, shared memory segment, exactly one of the following three values should be specified for **oflags**:

`O_RDONLY`	Open the shared memory special file for reading. The process can then map the shared memory special file with read access.
`O_WRONLY`	Open the shared memory special file for writing. The process can then map the shared memory special file with write access.

`O_RDWR` Open the shared memory special file for both reading and writing. The process can then map the shared memory special file with read and write access. If the shared memory file is persistent, the file will contain data placed in the memory represented by the file since `reboot`.

Where no data has been placed in the memory associated with the file, then the content of the memory is undefined. If the shared memory special file is not persistent, then on the first `open` the contents of the memory shall be undefined. After `reboot`, both the persistent and non-persistent shared memory special files are invalid and must be re-created.

Exceptions

For POSIX message queue special files, the following exceptions apply.

- If **`path`** refers to an existing, valid, message queue, only four flags are valid. Exactly one of the first three values should be specified for **`oflag`**:

  `O_RDONLY` Open the message queue special file for reading. The process can then use `mqreceive()`, or `mqgetevt()`, but not `mqsend()`, `mqputevt()`, or `mqpurge()`. A message queue can have multiple readers.

  `O_WRONLY` Open the message queue special file for writing. The process can then use `mqsend()`, `mqputevt()`, or `mqpurge()` on the queue, but not `mqreceive()` or `mqgetevt()`. A message queue can have multiple writers.

  `O_RDWR` Open the message queue special file for both reading and writing. The process can then use any of the functions allowed for `O_RDONLY` or for `O_WRONLY`.

  `O_NONBLOCK`
  Open the message queue special file in non-blocking mode: sends will not block due to

lack of resources and receives will not block waiting for messages.

Illegal values for **oflag** may not return an error, but may not produce the desired result. If the message queue special file is persistent, or if the file is already open, and if a system reboot has not occurred, then the message queue special file will contain messages that have been sent to the queue but not received.

If the message queue special file is not persistent, then on the first open the queue will be initialized to the empty state: the message queue special file will contain no messages.

- Note that on close of any message queue special file, any messages sent by the closing process with the MSG_USE flag are removed from the queue.
- Note that after a system reboot, the message queue special file is invalid and will need to be recreated using mkmq before it can be opened and used again. Any messages which were in the message queue before reboot will have been removed when it is recreated.

For STREAMS files, the following excetions apply.

- Only four flags O_RDONLY, O_WRONLY, O_RDWR and O_NDELAY have any meaning when opening a STREAMS file. The operation of STREAMS drivers and the system calls read(), write(), getmsg() and putmsg() are effected by O_NDELAY. The implementation of O_NDELAY is device specific for drivers.

DIAGNOSTICS

open returns the file descriptor number if successful, or -1 if not.

If open fails, **errno** will contain one of the following:

EACCES	Requested access for file denied according to the mode of the file, or a directory in the path denied search permission.
EEXIST	O_EXCL was requested, and the file exists.

EFAULT	`path` does not point to allocated memory in the process's address space.
EINTR	A signal interrupted the `open` call.
EISDIR	The file is a directory, and write access was requested.
ELOOP	Too many symbolic links were encountered when traversing the path.
EMFILE	The maximum number of files `open` per process has been reached.
ENAMETOOLONG	The path name is too long.
ENFILE	The system file table is full.
ENOENT	An element of the path prefix does not exist.
ENOENT	The file does not exist.
ENOSPC	The file system is out of inodes.
ENOSR	Cannot allocate a `stream`.
ENOTDIR	An element of the path prefix is not a directory.
ENXIO	The file is a device special file and the device does not exist.
ENXIO	`O_NDELAY` was requested, and no process had the FIFO file opened for reading.
ENXIO	A STREAMS module’s open failed.
EROFS	Write access is denied because file is on a read-only file system.
ETXTBSY	Write access is denied because the file is being executed.

SEE ALSO

System Calls - `chmod()`, `close()`, `dup()`, `fcntl()`, `lseek()`, `read()`, `umask()`, `write()`

Calls that operate on POSIX message queue special files include:

System Calls - `mkmq()`, `mqsend()`, `mqreceive()`, `mqsetattr()`, `mqgetattr()`, `mqpurge()`, `msgalloc()`, `msgfree()`, `mqputevt()`, `mqgetevt()`

Library Functions - `mqgetpid()`

Calls that operate on POSIX shared memory special files include:

System Calls - `mkshm()`, `shmmap()`, `shmunmap()`

The calls for STREAMS special files include:

System Calls - `getmsg()`, `putmsg()`, `poll()`

NAME

pipe – create a pipe

SYNOPSIS

```
pipe(fds)
int fds[2];
```

DESCRIPTION

pipe creates a first-in-first-out queue. The queue is accessed via two file descriptors; **fds**[0] is set to the file descriptor for the “read” end, and **fds**[1] is set to the file descriptor for the “write” end. Data written to the “write” end is readable on the “read” end.

To use a pipe for interprocess communication, a call to pipe should be made before a fork. The parent and child process now share both ends of the pipe. The process that wants to write into the pipe should close the “read” end with the close system call. The process that wants to read should close the “write” end of the pipe.

An end-of-file is returned to read calls on a pipe with no open write end. The SIGPIPE signal is sent to a process that tries to write to a pipe with no open “read” end.

The pipe buffer size is 4096 bytes. An attempt to write when the pipe is full causes the writing process to block until the pipe drains.

DIAGNOSTICS

pipe returns 0 if successful, or -1 if not.

If pipe fails, **errno** will contain one of the following:

EMFILE	The maximum number of files open per process has been reached.
ENOMEM	There is no more memory for pipes.
EFAULT	**fds** does not point to allocated memory in the process' address space.

NAME

`plock` – lock process in memory

SYNOPSIS

```
#include <sys/lock.h>

int plock()
```

DESCRIPTION

`plock` locks the calling process' text and data segments in memory. A locked process is immune to all swapping.

DIAGNOSTICS

`plock` returns 0 if the process was locked, or -1 if not.

If `plock` fails, **`errno`** will contain one of the following:

`ENOMEM`	There is not enough free memory in the system to lock the process.

SEE ALSO

System Calls - `execve()`, `exit()`, `fork()`

NAME

`poll` – input and output multiplexing on STREAMS

SYNOPSIS

```
#include <stropts.h>
#include <poll.h>

int poll(fds, nfds, timeout)
struct pollfd *fds;
unsigned long nfds;
int timeout;
```

DESCRIPTION

`poll` multiplexes input and output over a set of file descriptors which reference open `streams`. `poll` identifies `streams` on which a user can send or receive messages, or on which certain events occurred. A user can receive messages with `getmsg()` or `read()` system calls and send messages with `putmsg()` and `write()` system calls. In addition, some `ioctl` calls can be used to receive or send messages.

`fds` is a pointer to an array of **`pollfd`** structures. Each structure specifies a file descriptor to be examined and the events of interest for the file descriptor. **`pollfd`** structures contain the following members:

```
int fd;           /* file descriptor  */
short events;     /* requested events   */
short revents;     /* returned events   */
```

`fd` specifies an open file descriptor, and `events` and `revents` are bit masks which are created by OR-ing any combination of the following event flags:

`POLLIN` A non-priority message or a file descriptor passing message is present on the `stream head` read queue. This flag is set even when the message is zero length. This flag is mutually exclusive with `POLLPRI` in `revents`.

POLLPRI	A priority message present on the stream head read queue. This flag is set even if the message is zero length. This flag is mutually exclusive with POLLIN in revents.
POLLOUT	There is room in the first downstream write queue in the stream. Priority control messages can be sent at any time. This event and POLLHUP are mutually exclusive.
POLLERR	An error message has arrived at the stream head. This flag is only valid in the revents bit mask.
POLLHUP	A hangup occurred on the stream. This event and POLLOUT are mutually exclusive. This flag is only valid in the revents bit mask.
POLLNVAL	**fd** is not a file descriptor for an open stream. This flag is only valid in revents; it is not used in the events.

For each **pollfd** structure of the array, the file descriptor **fd** is examined by poll for events specified by events. The number of file descriptors examined is specified by **nfds**. If **nfds** exceeds the system limit of open files (NOFILES), poll will fail.

If **fd** is less than zero, events is ignored and revents is set to 0 when poll returns.

poll query results are stored in revents. Bits set in revents indicate which of the requested events are true. No bits are set when the poll call returns if none are true. POLLHUP, POLLERR, and POLLNVAL flags always set in revents, if the conditions they indicate are true.

In the event that no defined events occurred on a specified file descriptor, poll waits at least **timeout** msec. If **timeout** is 0, poll returns immediately. If **timeout** is -1, poll blocks until a requested event occurs, or the call is interrupted. poll is not affected by the O_NDELAY flag.

DIAGNOSTICS

The following values are returned:

non-negative value
Successful completion.

positive value	Indicates how many file descriptors (for which the `revents` field is non-zero) have been selected.
0	The `poll` call timed out; no file descriptors have been selected.
-1	System call failure; **errno** is set.

Possible values of **errno** are:

EAGAIN	Allocation of internal data structures failed; retry.
EFAULT	An argument points outside the allocated address space.
EINTR	A signal was caught during the `poll` call.
EINVAL	**nfds** is less than zero or greater than NOFILES.

SEE ALSO

System Calls - `getmsg(), putmsg(), read(), write()`

NAME

`profil` – execution time profile

SYNOPSIS

```
void profil(buff, bufsiz, offset, scale)
void(*offset) ();
char *buff;
int bufsiz;
int scale;
```

DESCRIPTION

`profil` is System V-compatible.

`profil` arranges for the operating system to examine the calling process' program counter on each clock tick, and create a histogram of the readings in the area of memory pointed to by **buff**. The size of **buff** (in bytes) is given by **bufsiz**. On each examination, **offset** is subtracted from the program counter, and the difference is multiplied by **scale**. If the resulting product corresponds to an entry in **buff**, that entry is incremented. Each entry consists of a short integer, so on most machines, the entry numbers range from 0 to **bufsiz**/2-1.

scale is interpreted as a 16-bit, unsigned, fixed-point fraction with binary point at the left. Thus, `0xffff` gives a 1-1 mapping of pc's to entries in **buff**; `0x7fff` gives a 2-1 mapping of pc's to entries in **buff**; etc. As a special case, a **scale** of `0x2` maps all pc values onto entry 0 in **buff** (producing a non-interrupting core clock).

Profiling is turned off by giving a **scale** of 0 or 1. It is rendered ineffective by giving a **bufsiz** of 0. Profiling is turned off when an `exec` is executed, but remains active in both the parent and child after a `fork`. Profiling is turned off if an update in **buff** would cause a memory fault.

DIAGNOSTICS

The return value from `profil` is undefined.

SEE ALSO

System Calls - `times()`

NAME

`ptrace` – process trace

SYNOPSIS

```
int ptrace(request, pid, addr, data)
int request;
int pid;
int addr;
int data;
```

DESCRIPTION

`ptrace` allows a process to control the execution of another process and its thread(s). `ptrace` also allows the controlling process to determine and modify various properties of the controlled process and its thread(s), such as registers, `USER` area, per-thread kernel data structure(s), data/heap segment, stack segment(s), and text segment. `ptrace` is usually used by a user program debugger. A controlled process is also referred herein to as a "traced process."

There are two ways for a process to start controlling another process: a process controls its direct descendent process; or a process controls an arbitrary process that may or may not be a direct descendent process of the controlling process.

In order for a process to control its direct descendent process (child process), the child process must call `ptrace` with **`request`** of `PTRACE_TRACEME`. This sets the "trace flag" in the child process' process data structure in the kernel space. The very next thing the child process normally would do is to call `execve` to load and transfer control to a new program. At this point, before the very first user instruction of the new program is executed, the child process stops as if it received a `SIGTRAP` signal. The parent (controlling) process should call `wait` so that it can know when the child process enters a stopped state.

In order for a process to control an arbitrary process, the controlling process must call `ptrace` with **`request`** of `PTRACE_ATTACH` and **`pid`** of the process ID of the process that the controlling process wishes to control. At this point, the traced process' "trace flag" is set and the traced process stops as if it received a `SIGTRAP` signal. The controlling process should call `wait` so that it can know when the child process enters a stopped state.

Once the controlling process has gained control of the traced process, it can apply any of the `ptrace` requests listed later on, except `PTRACE_TRACEME` and `PTRACE_ATTACH`, to the traced process. No matter how the tracing was initiated, `ptrace` requests will behave the same.

All `ptrace` requests except `PTRACE_TRACEME` and `PTRACE_ATTACH` must be made by the controlling process when the traced process is in a stopped state. The controlling process calls `wait` to know when the trace process enters a stopped state.

When a process is being traced, any signal that is sent to a thread in the traced process is trapped and the kernel forces all threads in the traced process, thus the entire process, to stop. The kernel preserves the original signal value in a thread data structure. The controlling process can retrieve the signal value as well as the thread ID of the thread to which the signal was sent through the `wait` call. The trapped signal is not automatically forwarded when the traced process is resumed. The controlling process must explicitly pass the signal, if needed, at its will. See `PTRACE_CONT`.

A number of special events also cause signals to be sent to threads in a traced process. These events include hitting breakpoints, single-stepping of execution, new threads being created, threads exiting, and system calls being issued (if `PTRACE_SYSCALL` is executed). These events are uniformly mapped to `SIGTRAP` in order to comply with UNIX convention; however, more precise analysis of the cause of a `SIGTRAP` signal is possible with `PTRACE_GETTRACESIG, Lynx's ptrace` extension.

A traced process is no longer traced when any of the following occurs:

1. The controlling process calls `ptrace` with **`request`** of `PTRACE_DETACH` and **`pid`** of the traced process' process ID;
2. The controlling process exits or is killed;
3. The traced process exits;

4. The traced process is killed by the controlling process by calling `ptrace` with **`request`** of `PTRACE_KILL` and **`pid`** of the traced process' process ID.

Each `ptrace` request applies either to the entire traced process (per-process) or to a single thread within the traced process (per-thread). Except `PTRACE_TRACEME`, all per-process requests take **`pid`** of a valid process ID of the traced process. All per-thread requests interpret **`pid`** as a combination of the process ID and thread ID of a particular thread in the traced process, built using the `BUILDPID` macro. For per-thread requests, if the thread ID portion of the combined process/thread ID is zero, the initial thread of the process will be chosen.

Since LynxOS is features concurrent multiple threads in a process and the LynxOS kernel is fully preemptable, the debuggers must be designed with special cautions regarding thread concurrency. One potential problem is that a breakpoint which was temporarily inserted or removed for a particular thread may be unexpectedly hit or ignored by another thread (see `PTRACE_SINGLESTEP_ONE`). Another is that when a thread hits a breakpoint, another thread may preempt the first one's execution, hit a breakpoint, and cause the traced process to stop, all before the first thread's exception handler delivers a signal to it; in this case, the debugger would first see only the second thread's stopping and the first thread's breakpoint signal (`SIGBREAKPOINT`) will be delivered after the traced process resumes to run; the debugger should not be confused even if the user removes the first thread's breakpoint before resuming the traced process.

The following is a list of valid `ptrace` **`request`** values supported by the LynxOS kernel:

`PTRACE_TRACEME` This request must be made by the child process in order for it to be controlled by its parent process. It sets the child's trace flag which will cause the child to enter a stopped state upon receipt of a signal rather than perform the normal action associated with the signal. The trace flag also causes the `execve` system call to send the calling process a `SIGTRAP` signal before executing the first instruction of the new program. Thus, if a child process calls `ptrace` with **`request`** `PTRACE_TRACEME` and then calls `execve`, it will enter a stopped state.

`ptrace` ignores the **`pid`**, **`addr`**, and **`data`** arguments and returns 0 on success. Per-process.

`PTRACE_ATTACH` This request attaches the process identified by **`pid`** and begins tracing it. `PTRACE_ATTACH` causes a `SIGSTOP` to be sent to the process. Process **`pid`** does not have to be a child process of the requesting process but the requesting process must have enough privilege to send process **`pid`** a signal. **`data`** and **`addr`** are ignored. Per-process.

`PTRACE_DETACH` This request detaches the process being traced. Process **`pid`** is no longer being traced and continues its execution. The **`data`** argument is taken as a signal number and the process continues as if it had incurred that signal. **`addr`** is ignored. Per-process.

`PTRACE_CONT` This request is used by the controlling process to cause the traced process to resume execution. The signal that caused the traced process to stop is always canceled. If a thread in the traced process has any other signals pending, the thread resumes execution by receiving one of those signals, which will cause the entire process to reenter the stopped state. Per-thread, but effective on the entire process.

A thread in the traced process can be made actually receive a signal without reentering a stopped state by setting **`data`** to a valid signal number and **`pid`** to a combination of the process ID and the thread ID of the thread (`BUILDPID`). This "special" signal is guaranteed to be delivered ahead of all other pending signals, if any, and is the only signal that does not cause the traced process to enter a stopped state. If **`data`** is 0, no special signal will be sent to the thread before it resumes execution. If the specified thread has a pending "special" signal (see below), the new value of **`data`** will override the pending signal value. The value of **`data`** is returned.

This request is always directed to an individual thread specified by **`pid`**, while all the threads in the traced process are also to be resumed. If the traced process contains more than one thread, the order of thread resumption is subject to the kernel's scheduling algorithm including thread priorities, and thus it is usually unknown which thread will resume first. Particularly, if a thread other than the one specified by **pid** resumes first and it causes the process to stop again due to signal reception or other reason, the specified thread may not have had a chance to run before the stopping at all. Also, in this case, the "special" signal passed in **data** becomes pending and will be delivered next time when the thread has a chance to run (scheduled after another resumption). Use PTRACE_CONT_ONE instead when this matters.

PTRACE_SINGLESTEP

This request allows the controlling process to single step a thread in the traced process by setting the trace bit in the Processor Status Word (PSW) of the thread and then letting the traced process resume execution as in PTRACE_CONT. The trace bit causes a processor exception after the specified thread executes one machine instruction. The corresponding exception handler sends SIGSINGLESTEP to the thread. A valid signal value in **data** can be used to deliver a signal to the specified thread as in PTRACE_CONT. Per-thread, but effective on the entire process.

On the x86 processor, the trace bit in the PSW corresponds to Trap Flag (TF) in the EFLAGS register, and the exception generated is interrupt 1. Single stepping of individual threads is allowed only on those architectures that support trace bits. On machines where the architectures do not support hardware single stepping, either the kernels or the debuggers must emulate single stepping by writing a breakpoint instruction at the next instruction location in the trace process' text area. Care must be taken: Any thread hitting such a breakpoint will stop, and the thread hitting the breakpoint may not be the one desired to hit the breakpoint.

If the traced process contains more than one thread, the specified thread will execute at most one machine instruction and causes the entire process to stop, while other threads in the same traced process are also resumed and may run continuously before the specified thread single steps. Since the order of thread resumption is subject to the kernel's scheduling algorithm including thread priorities, it is usually unknown which thread resumes first. So, it is possible that other threads than the specified one execute many instructions before the specified one single steps. Use `PTRACE_SINGLESTEP_ONE` instead when this matters.

`PTRACE_SYSCALL` The traced process is resumed as in `PTRACE_CONT`. When a thread in the traced process executes a system call, the traced process stops as if the thread had received a `SIGSYSCALL` signal. The controlling process can determine the system call number by peeking the appropriated field in the thread's data structure in the kernel space by `PTRACE_PEEKUSER`.

Per-thread, but effective on the entire process. *This feature is not yet implemented.*

`PTRACE_CONT_ONE`
`PTRACE_SINGLESTEP_ONE`
`PTRACE_SYSCALL_ONE`

These requests are single thread versions of `PTRACE_CONT` and `PTRACE_SINGLESTEP`, respectively. It is guaranteed that only the specified thread will resume while all other threads in the same process will remain stopped. Per-thread.

These request should be used only when a particular thread needs to be resumed exclusively in order to guarantee atomicity of a sequence of operations such as resumption from a breakpoint, or when the debugger provides a special function to the user. Otherwise, the non-`ONE` versions must be used to ensure concurrency of the threads in the traced process.

`PTRACE_SYSCAL_ONE` is not yet implemented.

`PTRACE_SETSIG` This request sets a "special" signal value **`data`** for the thread **`pid`**. The signal is guaranteed to be delivered when the thread is resumed to run. This request works similarly to `PTRACE_CONT` and `PTRACE_SINGLESTEP` only except that this request never actually resumes threads. Per-thread.

`PTRACE_GETTRACESIG`
This **`request`** returns the most recent signal that caused the traced process to stop. If the traced process stopped for one of the following reasons, `wait` indicates `SIGTRAP` but this **`request`** gives a more detailed cause of the stop; otherwise the same signal value (including `SIGTRAP`) will be returned. All other arguments than **`request`** are ignored. Per-thread.

The possible signal values (other than normal signals) returned are:

`SIGNEWTHREAD`
A new thread is created (but has not executed user code).

`SIGBREAKPOINT`
A thread hit a breakpoint.

`SIGSINGLESTEP`
A thread stopped due to single stepping.

`SIGTHREADEXIT`
A thread has exited.

`SIGSYSCALL`
A thread called a system call after resumed by `PTRACE_SYSCALL`. Not implemented.

`SIGEXECVE`
A thread executed the `execve` system call. Not implemented.

`PTRACE_KILL` This request is used by the controlling process to cause the traced process to terminate as if it had called `exit(0)`. Per-process.

`PTRACE_PEEKTEXT`
`PTRACE_PEEKDATA` These requests are used by the controlling process to read the address space of the traced process. The word at the location **`addr`** in the traced process' address space is returned. **`data`** is ignored. In LynxOS, there is no difference between the two requests. This is a per-process request since all threads within a process share the same virtual address space.

`PTRACE_POKETEXT`
`PTRACE_POKEDATA` These requests are used by the controlling process to write to the address space of the traced process. The word value of **`data`** is written to the location **`addr`** in the trace process' address space. The value of **`data`** is returned. In LynxOS, there is no difference between the two requests.

Since all threads within a process share the same virtual address space, writing to the address space of one thread makes a change to the address space of all the threads within the process. Therefore, this is a per-process request. Note that debuggers use this request to write breakpoint instructions into the text area of the process. Because the text area is shared by all threads within a process, all breakpoints are effective to all threads within the process.

`PTRACE_PEEKUSER` This request is used by the controlling process to read the `USER` area for the traced process. **`addr`** is an offset in the `USER` area in the operating system address space. The word at this offset in the `USER` area is returned. **`data`** is ignored. The `USER` area is a process-wide area that is shared by all threads within the process. The registers of a particular thread are not found in this area, but instead are found in the per-thread area. Per-process.

This request is maintained for backward compatibility and may become obsolete in a future release of LynxOS.

`PTRACE_POKEUSER` This request is used by the controlling process to write to a register of a thread in the traced process. **`addr`** is an offset in the per-thread data structure (`st_t`) in the kernel space. The word value of **data** is written at this offset in the thread data structure. The value of **data** is returned. Per-thread.

This request is maintained for backward compatibility and may become obsolete in a future release of LynxOS. Use PTRACE_SETREGS or PTRACE_SETFPREGS instead to alter register values.

PTRACE_PEEKTHREAD
: This request allows the controlling process to read the per-thread **data** pertaining to a thread in the traced process. The word at an offset **addr** in the thread's st_t structure is returned. Per-thread.

PTRACE_THREADUSER
: This request returns the address of the thread's st_t structure within the kernel space. **addr** and **data** are ignored.

PTRACE_GETREGS
: This request saves (reads out) the thread's regular registers into the user buffer pointer to **addr**. **data** is ignored. The structure of the regular register set is given in <proc.h>. Per-thread.

PTRACE_GETFPREGS
PTRACE_FPREAD
: These requests save (read out) the thread's floating point context into the user buffer pointer to by **addr**. **data** is ignored. The structure of the floating point context is given in <fpp.h>. Per-thread.

PTRACE_SETREGS
: This request restores (writes) the thread's regular registers from the user buffer pointed to by **addr**. **data** is ignored. The structure of the regular register set is given in <proc.h>. Per-thread.

PTRACE_SETFPREGS
PTRACE_FPWRITE
: These requests restore (write) the thread's floating point context from the user buffer pointed to by **addr**. **data** is ignored. The structure of the floating point context is given in <fpp.h>. Per-thread.

PT_READ_GRP	This request reads and returns the value of one of the thread's regular registers whose register number is **addr**. **data** is ignored. Per-thread. This request is currently available only to the PowerPC.
PT_WRITE_GRP	This request the word value of **data** to one of the thread's regular registers whose register number is **addr**. Per-thread. This request is currently available only to the PowerPC.
PT_READ_FRP PT_WRITE_FRP	These requests transfer a floating point value between the user buffer pointed to by **addr** and one of the thread's floating point registers whose register number is **data**. Per-thread. These requests are currently available only to the PowerPC.

DIAGNOSTICS

For requests PTRACE_GETTRACESIG, PTRACE_PEEKTEXT, PTRACE_PEEKDATA, PTRACE_PEEKUSER, PTRACE_POKETEXT, PTRACE_POKEDATA, PTRACE_POKEUSER, PTRACE_PEEKTHREAD, and PT_READ_GPR, ptrace returns the value read or written if successful.

For requests PTRACE_CONT, PTRACE_SINGLESTEP, PTRACE_CONT_ONE, PTRACE_SINGLESTE_ONE, and PTRACE_SETSIG, ptrace returns the value of **data** if successful, or -1 upon failure.

For the remaining requests, ptrace returns 0 if successful, or -1 upon failure.

If ptrace fails, errno will contain one of the following:

EIO	**request** is invalid.
EIO	**request** is PTRACE_PEEKTEXT, PTRACE_PEEKDATA, PTRACE_POKETEXT, or PTRACE_POKEDATA and **addr** does not point to allocated memory in the traced process's address space.
EIO	**request** is PTRACE_PEEKUSER or PTRACE_PEEKTHREAD and **addr** does not point to a valid location in the USER area of the thread.

EIO	**request** is PTRACE_POKEUSER and **addr** does not point to the valid location for a register.
EIO	**request** is PTRACE_CONT, PTRACE_SINGLESTEP, PTRACE_CONT_ONE, PTRACE_SINGLESTE_ONE, or PTRACE_SETSIG and **data** is not 0 or a valid signal number.
EIO	**request** is PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_SETREGS, PTRACE_SETFPREGS, PT_READ_GPR, PT_WRITE_GPR, PT_READ_FPR, or PT_WRITE_FPR, and the thread is created but has not executed the first user instruction (an embryo thread). The controlling process can check this by determining the value of ecp or the register stack buffer address in the thread's st_t structure with PTRACE_PEEKTHREAD, which is zero rather than some meaningful address value in the kernel's address space.
ESRCH	Process/thread **pid** either does not exist, is not stopped, has not called ptrace with **request** PTRACE_TRACEME, or has not been attached by PTRACE_ATTACH.

NAME

`putmsg` – send a message to a stream

SYNOPSIS

```
#include <stropts.h>

int putmsg(fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

DESCRIPTION

`putmsg` creates a message from user-specified buffers and sends it to a STREAMS file. `putmsg` builds the message from either a data part, a control part, or both that are passed to `putmsg` in separate buffers (see below). The STREAMS module that receives the message defines the semantics of each part.

The **`fd`** argument specifies a file descriptor referencing an open `stream.` **`ctlptr`** and **`dataptr`** each point to a **`strbuf`** structure which contains:

```
int  maxlen;  /* not used, see getmsg */
int  len;     /* length of data */
char  *buf;    /* ptr to buffer  */
```

`ctlptr` points to the structure specifying the control part which is to be included in the message, **`dataptr`** points to the structure specifying the data part. `buf` points to a buffer containing the control or data information. `len` indicates the number of bytes of data or control information to be sent. **`flags`** may be set to a value of 0 or `RS_HIPRI`.

`dataptr` must be non-NULL to send the data part of the message and `len` must have a value ≥ 0. The same rules apply to **`ctlptr`** in order to send the control part of the message. If **`dataptr/ctlptr`** is NULL or `len` is -1, no data/control part will be sent.

A priority message is sent when a control part is specified and **flags** is set to R_HIPRI. A non-priority message is sent if **flags** is set at 0. If **flags** is set to RS_HIPRI and no control part is specified, putmsg fails and sets **errno** to EINVAL. If **flags** is set at 0 and no data part and control part are specified, no message is sent; 0 is returned.

If the stream write queue is full due to internal flow control conditions, putmsg will block non-priority messages. If the write queue is full and O_NDELAY is set, putmsg does not block non-priority messages. Rather, it fails and sets **errno** to EAGAIN. Priority messages, however, are not blocked by putmsg if the stream write queue is full.

Regardless of priority or whether O_NDELAY is set, putmsg blocks waiting for the availability of message blocks in the stream; this is required so that a complete message can be sent.

DIAGNOSTICS

The following values are returned:

0	Successful completion.
-1	System call failure; **errno** is set.

Possible values of **errno** are:

EAGAIN	A non-priority message was specified; O_NDELAY is set; and the stream write queue is full due to internal flow control conditions, or buffers could not be allocated.
EBADF	**fd** is not a valid file decriptor.
EFAULT	**ctlptr** or **dataptr** points outside the allocated address space.
EINTR	A signal was caught during the putmsg system call.
EINVAL	An illegal value was specified in **flags**, or the **flags** is set to RS_HIPRI and no control part was supplied, or the stream referenced by **fd** is linked below a multiplexer.
ENOSTR	**fd** is not associated with a stream.

`ENXIO`	A hangup condition exists for the specified `stream`.
`ERANGE`	The size of the data part of the message is outside the limit specified by the min/max packet sizes of the uppermost `stream` module. `ERANGE` is also returned if the control part is larger than the range set for the control part of the message, or if the data part is larger than the maximum limit of the data part of the message.

In addition, `putmsg` will fail if a `STREAMS` error message is processed by the `stream head` before the call to `putmsg` takes place. The returned error represents the value contained in the `STREAMS` error message.

SEE ALSO

System Calls - `getmsg(), poll(), read(), write()`

NAME

`read, readv` – read from a file

SYNOPSIS

```
read(fd, buff, count)
int fd;
char *buff;
int count;

#include <uio.h>
readv(fd, iov, iovcnt)
int fd;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

`read` attempts to read **`count`** bytes from the open file descriptor **`fd`** into the buffer pointed to by **`buff`**. `readv` allows a single system call to distribute data among several different buffers. It scatters the input data into **`iovcnt`** buffers specified by the members of the **`iov`** array. Each **`iovec`** entry specifies the base address and length of a buffer:

```
struct iovec {
    caddr_t iov_base;  /* base address */
    int iov_len;       /* size in bytes */
};
```

If the file is a regular file or a device capable of seeking, then the seek pointer for the file is set to the file offset associated with **`fd`** and incremented as each character is read.

The actual number of characters read is returned. If the `seek` pointer is at end-of-file when `read` or `readv` is called, nothing is read, no error is generated, and a count of zero bytes is returned.

For STREAMS files, the following exceptions apply:

1. Three different modes (byte-stream, message-non-discard and message-discard) can be used to read from a STREAMS file.

The mode can be changed and tested using the `I_SRDOPT` and `I_GRDOPT` `ioctl` requests. The byte-stream mode is the default. In this mode `read` will read data from the `stream` until either **`count`** bytes have been read or the `stream` is empty. In byte-stream mode message boundaries are ignored. In the message-nondiscard mode `read` will return after reading **`count`** bytes or encountering a message boundary. If `read` does not encounter a message boundary then there is still data associated with the message on the `stream`. This data is left on the `stream` to retrieved by the next call to `read` or `getmsg`. The message-discard mode behaves like the message-nondiscard mode except that any data associated with a message that is not read from the `stream` is taken off of the `stream` and discarded.

2. Zero-byte messages are a special case that are handled differently by the byte-stream mode and the message based modes.

   If a zero-byte message is encountered in byte-stream mode, read returns the number of bytes read and the message is left on the `stream`. If in message-nondiscard or message-discard mode read returns 0 and removes the message from the `stream`. When reading from a STREAMS file only data messages can be read. Encountering a protocol message will cause `read` to fail.

The `O_NDELAY` flag effects different file types in different ways. The effect of the flag being set or cleared is outlined below:

1. When reading from an empty pipe `read` will return zero if `O_NDELAY` is set; otherwise, `read` will wait for data to appear or for the pipe to be closed.
2. When reading from a `tty` or `pty` that has no data available `read` will return zero if `O_NDELAY` is set; otherwise, `read` will wait for data to become available.
3. When reading from a STREAMS file that has no data available `read` will return -1 and set `errno` to `EAGAIN` if `O_NDELAY` is set; otherwise, `read` will wait for data to become available.

DIAGNOSTICS

When `read` or `readv` succeeds, the number of characters read is returned.

If `read` or `readv` fails, -1 is returned.

If the call fails, **`errno`** will contain one of the following:

`EAGAIN`	`O_NDELAY` is set and no data is available on the `stream`.
`EBADF`	**`fd`** is not a valid file descriptor.
`EBADMSG`	Attempted to read a message on a `stream` that is not a data message.
`EFAULT`	**`buff`** does not point to allocated memory in the process's address space.
`EFTYPE`	**`fd`** refers to a POSIX IPC special file.
`EINTR`	A signal interrupted the `read` call.
`EINVAL`	Tried to read from a `stream` linked to a multiplexer.
`ENXIO`	Invalid value for file pointer associated with the file descriptor of a special file.

SEE ALSO

System Calls - `dup(), fcntl(), flock(), ioctl(), open(), pipe(), execve()`

The calls for STREAMS special files include:

System Calls - `getmsg(), putmsg(), poll()`

NAME

readlink – read a symbolic link

SYNOPSIS

```
int readlink(path, buff, size)
char *path;
char *buff;
int size;
```

DESCRIPTION

readlink reads the symbolic link (a string) named **path** into the buffer pointed to by **buff**. The size of the buffer is given by **size**. The string will not be written past the end of the buffer, and will not be null-terminated.

readlink returns the length of the string found in the link.

DIAGNOSTICS

If readlink fails it returns -1.

If the call fails, **errno** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	Search permission is denied by some directory in the path.
ENXIO	The file is not a symbolic link.
EFAULT	**path** does not point to allocated memory in the process' address space.
EFAULT	**buff** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - `dup(), flock(), open(), pipe(), execve(), fcntl()`

NAME

`reboot` – reboot or halt the system

SYNOPSIS

```
#include <reboot.h>

int reboot(cmd)
int cmd;
```

DESCRIPTION

The `reboot` system call updates the reboot flags (either in non-volatile memory or on disk, depending upon the platform), and then if requested halts and reboots the system. **`cmd`** determines the function that is to be performed. **`cmd`** specifies the following:

`RB_ASKNAME`	Ask name of boot device before rebooting.
`RB_AUTOBOOT`	Skip single-user shell upon bootup.
`RB_NOTREALLY`	Do not reboot, but update the BBRAM.
`RB_NOFSCK`	Do file system check on bootup. (Yes, the name of this flag is confusing.)
`RB_PANIC`	Test panic.
`RB_HALT`	Halt the processor.
`RB_MAKENODES`	Tells `init` to remake nodes in `/dev` upon `reboot`.
`RB_TCACHE`	Mark cache invalid.
`RB_NOSYNC`	Do not flush cache before `reboot`.

Note that `reboot` can be called only by a process with effective user ID of 0 (super-user).

DIAGNOSTICS

If `reboot` is successful, and **`cmd`** is `RB_NOTREALLY`, a value of 0 is returned to the caller. In all other cases, when `reboot` is successful, it does not return to the caller.

If `reboot` fails, **`errno`** will be set to the following:

`EPERM`	Effective user ID of the process is not super-user.

SEE ALSO

System Calls - `shutdown()`

NAME

`recv`, `recvfrom` – receive a message from a socket

SYNOPSIS

```
#include <types.h>
#include <socket.h>

cc = recv(s, buf, len, flags)
int cc, s;
char *buf;
int len, flags;

cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;
```

DESCRIPTION

`recv` and `recvfrom` are used to receive messages from a socket.

The `recv` call may be used only on a connected socket (see `connect`), while `recvfrom` may be used to receive data on a socket whether it is in a connected state or not.

If **`from`** is non-zero, the source address of the message is filled in. **`fromlen`** is a value-result parameter, initialized to the size of the buffer associated with **`from`**, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in `cc`. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from; see `socket`.

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see `fcntl`) in which case a `cc` of `-1` is returned with the external variable **`errno`** set to `EWOULDBLOCK`.

The flags argument to a `recv` call is formed by the bitwise-OR of one or more of the values

```
#define MSG_PEEK 0x1 /* peek at incoming message */
#define MSG_OOB 0x2 /* process out-of-band data */
```

DIAGNOSTICS

`recv` and `recvfrom` return the number of bytes received if successful, or `-1` if not.

If the call fails, **`errno`** will contain one of the following:

`EBADF`	The argument **`s`** is an invalid descriptor.
`ENOTSOCK`	The argument **`s`** is not a socket.
`EWOULDBLOCK`	The socket is marked non-blocking and the receive operation would block.
`EINTR`	The receive was interrupted by delivery of a signal before any data was available for the receive.
`EFAULT`	The data was specified to be received into a non-existent or protected part of the process address space.

NOTE

For sockets of type `AF_UNIX`, `send` and `recv` have not been implemented. You must use `read` and `write` instead.

SEE ALSO

System Calls - `read(), send(), socket()`

NAME

`rename` – change the pathname of a file

SYNOPSIS

```
rename(path1, path2)
char *path1;
char *path2;
```

DESCRIPTION

`rename` makes a link named **path2** to the file named **path1**, and removes the link named **path1** all in one atomic operation. Both **path1** and **path2** must be on the same file system. **path1** must exist. If **path2** already exists, it will first be removed.

The implementation of rename ensures that at least one link to the file exists if the system crashes during a call.

DIAGNOSTICS

Rename returns 0 if the file was renamed, or -1 if not.

If the call fails, **errno** will contain one of the following:

ENOTDIR	An element of either path prefix is not a directory.
ENAMETOOLONG	**path1** or **path2** is too long.
ENOENT	An element of either path prefix does not exist.
ENOENT	**path1** does not exist.
ELOOP	Too many symbolic links were encountered when traversing one of the paths.
EROFS	Making the link requires writing to a directory on a read-only file system.
EACCES	Making the link requires writing to a directory that denies write permission.

`EFAULT`	**`path1`** or **`path2`** does not point to allocated memory in the process' address space.
`EXDEV`	**`path1`** and **`path2`** are on different file systems.

SEE ALSO

System Calls - `link()`

NAME

`rmdir` – remove a directory

SYNOPSIS

```
rmdir(path)
char *path;
```

DESCRIPTION

`rmdir` removes a directory file named **path**. The directory must be empty except for "`.`" and "`..`".

DIAGNOSTICS

`rmdir` returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain one of the following:

ENOTDIR	An element of the **path** is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The directory does not exist.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	Write access for directory denied according to the mode of the file, or a directory in the **path** denied search permission.
EFAULT	**path** does not point to allocated memory in the process' address space.
EBUSY	The directory is the root of a mounted system.
EROFS	The directory is on a read-only file system.
ENOTEMPTY	The directory has entries other than just "`.`" and "`..`".

SEE ALSO

System Calls - `mkdir()`, `unlink()`

NAME

select – I/O multiplexing

SYNOPSIS

```
#include <types.h>
#include <time.h>

int select(nfds, readfds, writefds, exceptfds, timeout)
int nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
FD_SET(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

select allows a process to wait for the condition of one of several open file descriptors to change. The change may be in the form of an exceptional condition, or it may be entry into a state in which an I/O operation will not block (i.e., available data for an input descriptor or available buffer space for an output descriptor). The arguments **readfds**, **writefds**, and **exceptfds** point to structures which describe sets of file descriptors. These structures may be long integers in which case 32 file descriptors could be monitored. If more than 32 files descriptors are required, pointers to structures with at least **nfds** bits can be passed instead. If a particular file descriptor (numbered 0-**nfds**) is to be included in one of the sets, the bit corresponding to the file descriptor is set to one. Only the first **nfds** (low order) bits of each mask are recognized.

File descriptors referenced in **readfds** will be monitored for available input data. Similarly, file descriptors in **writefds** will be monitored for availability of output buffer space or device readiness. File descriptors in **exceptfds** will be monitored for appearance of some exceptional condition. A single file descriptor may be mentioned in any or all of the sets.

The argument **timeout** specifies the maximum time to wait for something to happen to any of the relevant descriptors. If **timeout** is NULL, the wait is indefinite. If the process does not wish to wait at all, **timeout** should point to a **timeval** structure with zero value.

`select` returns the number of ready file descriptors. The sets are modified to contain only those descriptors which have been affected. Note that due to interaction between separate processes, `select` may report that some condition exists that no longer exists by the time an actual I/O operation is performed.

The descriptor sets are stored in **fd_set** data type, an array of long integers. The following macros are provided in `/usr/include/types.h` to use the descriptor sets:

`FD_SET(fd, &fdset)`
: set the bit for file descriptor **fd** in **fdset**.

`FD-CLR(fd &*fdset)`
: clear the bit for file descriptor **fd** in **fdset**

`FD_ZERO(&fdset)`
: initialize all bits in **fdset**

`FD_ISSET(fd &fdset)`
: see if file descriptor **fd** is in the set. The value is nonzero if **fd** is a member of **fdset**; zero otherwise.

DIAGNOSTICS

`select` returns `-1` if it fails.

If the call fails, **errno** will contain one of the following:

EBADF	One of the sets contains an invalid file descriptor.
EFAULT	**readfds**, **writefds**, or **exceptfds** does not point to allocated memory in the process' address space.
EFTYPE	**fd** refers to a POSIX `ipc` special file.
EINTR	A signal was received before any of the file descriptors were ready.

NOTE

Whether or not `select` successfully detects a status change on a descriptor before returning, the contents of the file descriptor sets are changed.

SEE ALSO

System Calls - `read()`, `write()`

NAME

`sem_count` – get a user semaphore count

SYNOPSIS

```
int semcount(sd)
int sd;
```

DESCRIPTION

`sem_count` returns the count of the user semaphore identified by **sd**.

DIAGNOSTICS

`sem_count` returns -1 on error, but because -1 is a legal semaphore count, to check for an error, **errno** should be cleared before the call.

If the call fails, **errno** will contain the following:

EBADF **sd** is not a valid semaphore descriptor.

NOTES

You must explicitly link with the library `liblynx.a` to get this system call.

Count is negative is there are no waiters.

SEE ALSO

System Calls - `sem_get()`, `sem_wait()`, `sem_signal()`, `sem_reset()`, `sem_delete()`

NAME

semctl – semaphore control operations

SYNOPSIS

```
#include <ipc.h>
#include <sem.h>

int semctl (semid, semnum, cmd, arg)
int semid;
int semnum;
int cmd;
union semun {
int val;
struct semid_ds *buff;
unsigned short *array;
} arg;
```

DESCRIPTION

`semctl` is System V-compatible.

`semctl` provides various controls over a System V-compatible semaphore or set of semaphores. Some of the operations given by **cmd** act on a semaphore set and others act on one particular semaphore within a set. The operations that act over the semaphore set specified by **semid** are given by the following values for **cmd**:

GETALL	Read the values of every semaphore in the set specified by **semid** into the array pointed to by **arg.array**.
SETALL	Set the values of every semaphore in the set specified by **semid** according to the array pointed to by **arg.array**. In addition, zero out each process' `semadj` values for each semaphore in the set.
IPC_STAT	Populate the fields of the structure pointed to by **arg.buff** with the data in the `semid_ds` structure associated with **semid**.

IPC_SET Populate the following fields of the semid_ds structure associated with **semid** with the corresponding fields of the structure pointed to by **arg.buff**:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode (low-order 9 bits)
```

IPC_RMID Remove the semaphore set specified by **semid**.

The IPC_SET and IPC_RMID commands can only be executed by a process with an effective user ID equal either to super-user or to the value of sem_perm.cuid or sem_perm.uid in the semid_ds structure associated with **semid**.

The semid_ds structure is defined in sem.h as follows:

```
struct semid_ds {
  struct ipc_perm sem_perm;  /* access permissions */
  char *sem_base;  /* pointer to first sem.(useless) */
  unsigned short sem_nsems;  /* number of semaphores  */
  unsigned long sem_otime;  /* time of last operation */
  unsigned long sem_ctime;  /* time of last change */
};
```

The following fields are part of the data structure associated with each individual semaphore:

```
unsigned semval; /* semaphore value */
int semncnt;     /* # procs waiting for semval  > curval */
int semzcnt;     /* # procs waiting for semval  = 0 */
int sempid;      /* pid of proc that did last operation *
```

The operations that act over the semaphore specified by **semnum** within the semaphore set specified by semid are given by the following values for **cmd**:

GETVAL Return the value of semval.

SETVAL Set the value of semval to **arg.val** and zero out in each process the semadj value for the specified semaphore.

GETPID Return the value of sempid.

GETNCNT	Return the value of semncnt.
GETZCNT	Return the value of semzcnt.

DIAGNOSTICS

semctl returns 0 if successful, or -1 if not.

If semctl fails, **errno** will contain one of the following:

EINVAL	**semid** is not a valid semaphore set identifier.
EINVAL	**semnum** is less than zero or greater than sem_nsems.
EINVAL	**cmd** is not a valid command.
EACCES	Operation permission is denied to the calling process.
ERANGE	**cmd** is equal to SETVAL or SETALL and semval is to be set greater than the system limit.
EPERM	**cmd** is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is neither super-user nor equal to sem_perm.cuid or sem_perm.uid in the semid_ds structure associated with **semid**.
EFAULT	**arg.buff** points to an illegal address.

NAME

`sem_delete` – delete a user semaphore

SYNOPSIS

```
sem_delete(name)
char *name;
```

DESCRIPTION

`sem_delete` frees up the system entry for the semaphore identified by name and removes it from the list of active semaphores.

DIAGNOSTICS

`sem_delete` returns 0 if the semaphore was deleted, or -1 if not.

If the call fail, **errno** will contain one of the following:

EFAULT	`name` points outside the user's address space.
EBUSY	Some process is waiting on the semaphore (the count is negative).
EINVAL	No semaphore exists with `name`.

NOTES

You must explicitly link with the library `liblynx.a` to get this system call.

SEE ALSO

System Calls - `sem_get()`, `sem_wait()`, `sem_signal()`, `sem_reset()`, `sem_count()`

NAME

sem_get – get a user semaphore

SYNOPSIS

```
int sem_get(name, count)
char *name;
int count;
```

DESCRIPTION

sem_get returns a semaphore descriptor for the semaphore **name**. If the semaphore exists, its descriptor is returned; if not, a new semaphore is created with an initial count of **count**.

The name of a semaphore is an arbitrary null-terminated string. It has nothing to do with the file system or the shared memory system.

As a special case, if **count** is –1, and the semaphore is being created, then priority inheritance will be performed on the new semaphore, and the semaphore will be initialized in the unlocked state (equivalent to an initial count of 1). Briefly, priority inheritance means that any process owning the semaphore will run at a priority equal to the maximum of its own priority and the priorities of all processes currently blocked on the semaphore.

Except for the special case of –1, **count** must not be negative.

DIAGNOSTICS

sem_get returns -1 if it fails to get the semaphore.

If the call fails, **errno** will contain one of the following:

EFAULT	**name** does not point to allocated memory in the process' address space.
EINVAL	**count** is negative and not equal to –1.

NOTES

You must explicitly link with the library liblynx.a to get this system call.

SEE ALSO

System Calls - `sem_delete()`, `sem_wait()`, `sem_signal()`, `sem_reset()`, `sem_count()`

NAME

semget – get semaphore set

SYNOPSIS

```
#include <ipc.h>
#include <sem.h>

int semget(key, nsems, flag)
int key;
int nsems;
int flag;
```

DESCRIPTION

semget is System V-compatible.

semget returns the semaphore set identifier associated with **key**. If the semaphore set exists, its identifier is returned; if not, a new semaphore set is created as long as the IPC_CREAT bit is set in **flag**.

A new semaphore set will always be created if **key** is equal to IPC_PRIVATE.

When a new semaphore set is created, its associated semid_ds data structure is initialized as follows:

sem_perm.cuid	set equal to the effective user ID.
sem_perm.uid	set equal to the effective user ID.
sem_perm.cgid	set equal to the effective group ID.
sem_perm.gid	set equal to the effective group ID.
sem_perm.mode	low-order 9 bits set equal to low-order 9 bits of **flag**.
sem_nsems	set equal to **nsems**.
sem_otime	set equal to zero.
sem_ctime	set equal to the current time.

Each semaphore in the set has an associated `sem` data structure that is not initialized. Each semaphore can be initialized using the `semctl` system call with the command `SETVAL` or `SETALL`.

DIAGNOSTICS

`semget` returns the semaphore set identifier if successful, or `-1` if not.

If `semget` fails, **`errno`** will contain one of the following:

`EINVAL`	**`nsems`** is either less than or equal to zero or greater than the system limit.
`EINVAL`	A semaphore set exists for **`key`**, but the number of semaphores in the set is less than **`nsems`**.
`EACCES`	A semaphore set exists for **`key`**, but operation permission is denied according to the low-order 9 bits of **`flag`**.
`ENOENT`	No semaphore set exists for **`key`** and the `IPC_CREAT` bit is not set in **`flag`**.
`ENOSPC`	A semaphore set is to be created, but there are no more semaphore set identifiers.
`EEXIST`	A semaphore set exists for **`key`** but the `IPC_CREAT` and `IPC_EXCL` bits are both set in **`flag`**.

NAME

semop – semaphore operations

SYNOPSIS

```
#include <ipc.h>
#include <sem.h>

int semop(semid, sops, nsops)
int semid;
struct sembuf *sops;
unsigned nsops;
```

DESCRIPTION

semop is System V-compatible.

semop performs a series of semaphore operations on the set of semaphores associated with identifier **semid**. The operations are contained in an array of structures pointed to by **sops**. The number of semaphore operation structures in the array is given by **nsops**. The semaphore operation structure is defined in sem.h as follows:

```
struct sembuf {
  short   sem_num;        /* number of semaphore */
  short   sem_op;         /* operation */
  short   sem_flg;        /* ipc flag */
};
```

sem_num specifies the semaphore in the set associated with **semid** on which to perform the operation specified by sem_op.

sem_op can specify one of three possible semaphore operations, depending on whether it is negative, positive, or zero:

If sem_op is negative, one of the following occurs:

- If semval is greater than or equal to the absolute value of sem_op, the absolute value of sem_op is subtracted from semval. In addition, if the SEM_UNDO bit is set in sem_flg, the absolute value of sem_op is added to the calling process' semadj for this semaphore.

- If semval is less than the absolute value of sem_op and the IPC_NOWAIT bit is set in sem_flg, the semop call fails with a return value of -1 and **errno** set to EAGAIN.
- If semval is less than the absolute value of sem_op and the IPC_NOWAIT bit is not set in sem_flg, semncnt in the sem structure associated with the semaphore is incremented and the process is suspended until one of the following occurs:
  - semval becomes greater than or equal to the absolute value of sem_op. In this case, semncnt in the sem structure associated with the semaphore is decremented, the absolute value of sem_op is subtracted from semval, and, if the SEM_UNDO bit is set in sem_flg, the absolute value of sem_op is added to the calling process semadj value for this semaphore.
  - The semaphore set associated with **semid** is removed from the system, in which case a value of -1 is returned with **errno** set to EIDRM.
  - The process receives a signal that is to be caught, in which case semncnt in the sem structure associated semaphore is decremented and a value of - 1 is returned with **errno** set to EINTR.
- If sem_op is positive, sem_op is added to semval and, if the SEM_UNDO bit is set in sem_flg, sem_op is subtracted from the calling process' semadj value for this semaphore.
- If sem_op is zero, one of the following occurs:
  - If semval is zero, the semop call returns immediately.
  - If semval is not zero and the IPC_NOWAIT bit is set in sem_flg, the semop call fails with a return value of -1 and **errno** set to EAGAIN.
  - If semval is not zero and the IPC_NOWAIT bit is not set in sem_flg, semzcnt in the sem structure associated with the semaphore is incremented and the process is suspended until one of the following occurs:
    * semval becomes zero, in which case semzcnt in the sem structure associated with the semaphore is decremented.

- * The semaphore set associated with **semid** is removed from the system, in which case a value of -1 is returned with **errno** set to EIDRM.
- * The process receives a signal that is to be caught, in which case semzcnt in the sem structure associated semaphore is decremented and a value of -1 is returned with **errno** set to EINTR.

- If semop succeeds, sempid in the sem structure associated with each semaphore specified in the array pointed to by **sops** is set equal to the process ID of the calling process.

DIAGNOSTICS

semop returns 0 if successful, or -1 if not. If semop fails, **errno** will contain one of the following:

EINVAL	**semid** is not a valid semaphore set identifier.
EINVAL	The number of semaphores for which the calling process requests a SEM_UNDO would exceed the limit.
EFBIG	sem_num is less than zero or greater than or equal to the number of semaphores in the set associated with **semid**.
E2BIG	**nsops** is greater than the system limit.
EACCES	Operation permission is denied to the calling process.
EAGAIN	The operation would suspend the calling process but the IPC_NOWAIT bit is set in sem_flg.
ENOSPC	The number of processes requesting a SEM_UNDO would exceed the system limit.
ERANGE	An operation would cause a semval to overflow the system limit.
ERANGE	An operation would cause a semadj to overflow the system limit.
EFAULT	**sops** points to an illegal address.

`EIDRM`	The calling process was suspended and the semaphore set associated with **`semid`** is removed from the system.
`EINTR`	The calling process was suspended and it receives a signal that is to be caught.

NAME

`sempost, semifpost` – post to a binary semaphore (POSIX.4, Draft 9)

SYNOPSIS

```
int sempost(fildes)
int fildes;

int semifpost(fildes)
int fildes;
```

DESCRIPTION

`sempost` performs the V operation on the file descriptor **`fildes`**. `sempost` changes the semaphore state to `SEM_UNLOCKED` if the semaphore is `SEM_LOCKED` and no processes are blocked waiting for the semaphore. If the semaphore is `SEM_LOCKED` and there are processes blocked waiting for the semaphore, the state of the semaphore does not change. If the semaphore is `SEM_UNLOCKED`, the state of the semaphore does not change.

`semifpost` performs the V operation on **`fildes`** only if there are processes blocked waiting for the semaphore. If the semaphore is locked but no processes are waiting for the semaphore, `semifpost` fails and returns an error. If processes are waiting for the semaphore, `semifpost` returns successfully and the state of the semaphore is unchanged.

If more than one process is blocked waiting for a semaphore, the highest priority process that has been waiting the longest is unblocked.

DIAGNOSTICS

`sempost` and `semifpost` unlock the binary semaphore and return 0 if successful. Otherwise, the state of the binary semaphore will remain unchanged, a value of `-1` is returned, and **`errno`** is set to one of the following values:

`EAGAIN`	No process waiting on the binary semaphore (`semifpost` only).
`EBADF`	**`fildes`** is not a valid file descriptor.
`EFTYPE`	**`fildes`** is not a binary semaphore.

CONFORMANCE

`sempost` and `semifpost` are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `close()`, `open()`, `mksem()`, `sempost()`, `semifpost()`

Example - `semifpost.c`

NAME

`sem_reset` – reset a user semaphore

SYNOPSIS

```
sem_reset(sd)
int sd;
```

DESCRIPTION

`sem_reset` wakes all processes waiting on the semaphore given by **sd** and sets the semaphore count to 0.

DIAGNOSTICS

`sem_reset` returns 0 if successful, -1 if not.

If the call fails, **errno** will contain the following:

EBADF	**sd** is not a valid semaphore descriptor.

NOTES

You must explicitly link with the library `liblynx.a` to get this system call.

SEE ALSO

System Calls - `sem_get(), sem_wait(), sem_signal(), sem_count(), sem_delete()`

NAME

sem_signal, sem_nsignal – signal a user semaphore

SYNOPSIS

```
sem_signal(sd)
int sd;

sem_nsignal(sd, n)
int sd;
int n;
```

DESCRIPTION

sem_signal increments the semaphore count for semaphore **sd**. If there are any processes waiting on the semaphore, the first one in the queue is awaken. That process then returns from its sem_wait call.

sem_nsignal signals the semaphore **n** times.

These calls have nothing to do with the signal facilities of the operating system. Sending a signal to a process and signaling a semaphore are two completely different things.

DIAGNOSTICS

Both sem_signal and sem_nsignal return 0 if successful, or -1 if not.

If the call fails, **errno** will contain one of the following:

EBADF	**sd** is not a valid semaphore descriptor.
EINVAL	count is negative.

NOTES

You must explicitly link with the library liblynx.a to get this system call.

SEE ALSO

System Calls - `sem_get(), sem_wait(), sem_reset(), sem_count(), sem_delete()`

NAME

sem_wait – wait on a user semaphore

SYNOPSIS

```
sem_wait(sd)
int sd;
```

DESCRIPTION

sem_wait decrements the semaphore count for the semaphore given by **sd** and returns immediately if the count is non-negative. If the count is negative after the decrement performed by the call, the calling process is placed in the queue of processes waiting on the semaphore and sleeps. The wait queues are kept in priority (then FIFO within each priority) order.

After the calling process returns from sem_wait, it is the “owner” of whatever resource the semaphore arbitrates. The process must signal the semaphore with sem_signal to give up that ownership.

DIAGNOSTICS

sem_wait returns 0 if the wait succeeded, or -1 if not.

If the call fails, **errno** will contain the following:

EBADF	**sd** is not a valid semaphore descriptor.

NOTES

You must explicitly link with the library liblynx.a to get this system call.

SEE ALSO

System Calls - sem_get(), sem_reset(), sem_signal(), sem_count(), sem_delete()

NAME

`semwait, semifwait` – wait on a binary semaphore (POSIX.4, Draft 9)

SYNOPSIS

```
int semwait(fildes)
int fildes;

int semifwait(fildes)
int fildes;
```

DESCRIPTION

`semwait` locks the binary semaphore specified by **`fildes`** if the semaphore is unlocked. **`fildes`** should be the return value of a successful open of a binary semaphore special file. If the semaphore is in the `SEM_LOCKED` state, the function will block until the semaphore is unlocked (see `sempost`). When `semwait` returns successfully, the state of the semaphore will be `SEM_LOCKED`.

`semifwait` locks the binary semaphore only if the semaphore is unlocked. If the state of the semaphore is `SEM_LOCKED`, `semifwait` will fail and return an error. If the state of the semaphore is `SEM_UNLOCKED`, `semifwait` will return successfully and the state of the semaphore will be `SEM_LOCKED`.

Signals or events sent to a process blocked on `semifwait` will cause the function to return with an error.

DIAGNOSTICS

`semwait` and `semifwait` lock the binary semaphore and return 0 if successful. Otherwise, the state of the binary semaphore will remain unchanged, a value of -1 is returned, and **`errno`** is set to one of the following values:

EAGAIN	Semaphore state is `SEM_LOCKED` (`semifwait` only).
EBADF	**`fildes`** is not a valid file descriptor.
EFTYPE	**`fildes`** is not a binary semaphore.

`EINTR`	An event or signal interrupted the function.

CONFORMANCE

`semwait` and `semifwait` are implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`close()`, `open()`, `mksem()`, `sempost()`, `semifpost()`
Example -	`semwait.c`, `semifwait.c`

NAME

send, sendto – send a message from a socket

SYNOPSIS

```
#include <types.h>
#include <socket.h>

cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;

cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;
```

DESCRIPTION

send and sendto are used to transmit a message to another socket. send may be used only when the socket is in a connected state, while sendto may be used at any time.

The address of the target is given by **to** with **tolen** specifying its size. The length of the message is given by **len**. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send. Return values of –1 indicate some locally-detected errors.

If no message space is available at the socket to hold the message to be transmitted, then send normally blocks, unless the socket has been placed in non-blocking I/O mode.

The **flags** parameter may be set to SOF_OOB to send "out-of-band" data on sockets which support this notion (e.g., SOCK_STREAM). See recv for a description of the msghdr structure.

DIAGNOSTICS

`send` and `sendto` return the number of characters sent if successful, or `-1` if not.

If the call fails, **`errno`** will contain one of the following:

EBADF	An invalid descriptor was specified.
ENOTSOCK	The argument **s** is not a socket.
EFAULT	An invalid user space address was specified for a parameter.
EMSGSIZE	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.

NOTE

For sockets of type `AF_UNIX`, `send` and `recv` have not been implemented. You must use `read` and `write` instead.

SEE ALSO

System Calls - `recv(), socket(), write()`

NAME

`setpgid` – set process group ID for job control

SYNOPSIS

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

int setpgid(pid, pgid)
pid_t pid;
pid_t pgid);
```

DESCRIPTION

The `setpgid()` function is used to either join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader shall not change. Upon successful completion, the process group ID of the process with a process ID that matches **pid** is set to **pgid**. As a special case, if **pid** is zero, the process ID of the calling process is used. Also, if **pgid** is zero, the process ID of the indicated process is used.

DIAGNOSTICS

Upon successful completion, `setpgid()` returns a value of zero. Otherwise, a value of `-1` is returned and **errno** is set to indicate the error.

If any of the following conditions occur, the `setpgid()` function shall return `-1` and set **errno** to the corresponding value:

EACCES	The value of the **pid** argument matches the process ID of a child process of the calling process, and the child process has successfully executed one of the `exec` functions.
EINVAL	The value of the **pgid** argument is less than zero or is not a value supported by the implementation.
ENOSYS	The `setpgid()` function is not supported in this implementation.
EPERM	The process indicated by the **pid** argument is a session leader.

The value of the **pid** argument is valid, but matches the process ID of a child process of the calling process, and the child process is not in the same session as the calling process.

The value of the **pgid** argument does not match the process ID of the process indicated by the **pid** argument, and there is no process with a process group ID that matches the value of the **pgid** argument in the same session as the calling process.

`ESRCH`	The value of the **pid** argument does not match the process ID of the calling process or of a child process of the calling process.

COPYRIGHT NOTICE

SEE ALSO

System Calls - `execve(), getpgrp(), setsid()`

Library Functions - `execl(), tcsetpgrp()`

NAME

`setprio` – set process priority

SYNOPSIS

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

int setprio (pid, prio)
pid_t pid;
int prio;
```

DESCRIPTION

`setprio` sets the priority of the process specified by **`pid`** to **`prio`**. The value of **`prio`** shall be any integer in the range `PRIO_MIN` and `PRIO_MAX` as defined for the current scheduling policy. If **`pid`** is 0, the priority shall be set for the calling process.

For a process to have permission to set the priority of a process, it must either have root privileges, or the real or effective user ID of the calling process must match the real or effective user ID of the target process.

The target process shall resume execution after all other runnable processes of equal or greater priority have been scheduled to run. The priority of a process is inherited across `fork` and `exec`.

If the priority of the target process is set higher than that of the calling process, and if the target process is ready to run, the target process will preempt the calling process. If the calling process sets its priority lower than that of the target process, and if the target process is ready to run, the target process shall preempt the calling process.

DIAGNOSTICS

`setprio` returns old priority value if successful, or -1 if not.

If `setprio` fails, **`errno`** will contain one of the following:

`EINVAL`	**`prio`** is outside the range of priorities for the current scheduling algorithm.

`EPERM`	The effective or real **`uids`** for process **`pid`** do not match the effective or real **`uids`** for the calling process, and the calling process is not privileged.
`EPERM`	**`prio`** is greater than the maximum non-privileged priority, and the caller is non-privileged.
`ESRCH`	**`pid`** does not specify a valid process.

CONFORMANCE

`setprio` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`getpid(), getprio(), getscheduler(), setscheduler()`
Examples -	`setprio.c`

NAME

`setreuid`, `setregid` – set real and effective ID

SYNOPSIS

```
setreuid(uid, euid)
int uid, euid;

setregid(gid, egid)
int gid, egid;
```

DESCRIPTION

The routines `setreuid` and `setregid` set the real and effective user or group ID values for the current process. The effective user and group ID values are used for resource access checks within the operating system, and by various utility programs. The real ID values are used by some utilities to ensure that privilege for a process existed before executing the program, which might be installed with the `setuid` bit enabled.

Only privileged processes can change the real ID or change the effective ID to a value different from the real ID. Other processes can only set the effective ID to the real ID, if it is different.

User and group ID values must lie in the range 0 to 65534. If only one of the two values need be set, the other should be given as -1.

DIAGNOSTICS

These routines return -1 if an error is detected, or 0 if not. If an error occurs, the global variable **`errno`** will contain one of the following:

`EPERM`	The process attempted to change a real ID or to change the effective ID to a value different from the real ID, and the process was not a super-user process.
`EINVAL`	Either the real or effective ID requested was greater than 65534 or less than 0 (and not -1).

SEE ALSO

System Calls - `getuid(), getgid()`

NAME

`setscheduler` – set process scheduling policy and priority (POSIX.4, Draft 9)

SYNOPSIS

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

int setscheduler(pid, alg, prio)
pid_t pid;
int alg;
int prio;
```

DESCRIPTION

`setscheduler` sets the scheduling policy and priority of the process specified by **pid** to **prio**. The value of **prio** shall be any integer in the range `PRIO_MIN` and `PRIO_MAX` as defined for the scheduling policy specified by **alg**. If **pid** is 0, the priority and scheduling policy shall be set for the calling process.

The value of **alg** may be one of the following:

`SCHED_FIFO`	A first in-first out (FIFO) scheduling policy.
`SCHED_RR`	A round robin scheduling policy.
`SCHED_DEFAULT`	A priority-based quantum scheduling policy.
`SCHED_OTHER`	Identical to `SCHED_DEFAULT`

For a process to have permission to set the priority or scheduling policy of a process, it must either have root privileges, or the real or effective user ID of the calling process must match the real or effective user ID of the target process.

The target process shall resume execution after all other runnable processes of equal or greater priority have been scheduled to run. The priority of a process and its scheduling policy is inherited across `fork` and `exec`.

If the priority of the target process is set higher than that of the calling process, and if the target process is ready to run, the target process will preempt the calling process. If the calling process sets its priority lower than that of the target process, and if the target process is ready to run, the target process shall preempt the calling process.

DIAGNOSTICS

`setscheduler` returns old scheduling policy if successful, or -1 if not.

If setscheduler fails, **errno** will contain one of the following:

EINVAL	**prio** is outside the range of priorities for the specified scheduling policy, or **alg** is not a valid scheduling policy.
EPERM	The effective or real **uids** for process **pid** do not match the effective or real **uids** for the calling process, and the calling process is not privileged.
EPERM	**prio** is greater than the maximum non-privileged priority, and the caller is non-privileged.
ESRCH	**pid** does not specify a valid process.

CONFORMANCE

`setscheduler` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls -	`getpid(), getprio(), getscheduler(), setprio()`
Example -	`setscheduler.c`

NAME

`setsid` – create session and set process group ID

SYNOPSIS

```
#include <sys/types.h>
pid_t setsid();
```

DESCRIPTION

If the calling process is not a process group leader, the `setsid()` function creates a new session. The calling process will be the session leader of this new session, will be the process group leader of a new process group, and shall have no controlling terminal. The process group ID of the calling process is set equal to the process ID of the calling process. The calling process will be the only process in the new process group and the only process in the new session.

DIAGNOSTICS

Upon successful completion, the `setsid()` function returns the value of the process group ID of the calling process. Otherwise, a value of `-1` is returned and **`errno`** is set to indicate the error.

If any of the following conditions occur, the `setsid()` function returns `-1` and sets **`errno`** to the corresponding value:

`EPERM`	The calling process is already a process group leader or the process group ID of a process other than the calling process matches the process ID of the calling process.

COPYRIGHT NOTICE

misinterpretation of said information resulting from the placement and context in this publication. Information is reproduced with the permission of the IEEE.

SEE ALSO

System Calls - `setpgid(), sigaction()`

NAME

`shmctl` – shared memory control operations

SYNOPSIS

```
#include <ipc.h>
#include <shm.h>

int shmctl(shmid, cmd, buff)
int shmid;
int cmd;
struct shmid_ds *buff;
```

DESCRIPTION

`shmctl` is System V-compatible.

`shmctl` provides various controls over a System V-compatible shared memory segment. The operations are given by the following values for **`cmd`**:

`IPC_STAT`	Populate the fields of the structure pointed to by **`buff`** with the data in the `shmid_ds` structure associated with **`shmid`**.

`IPC_SET`	Populate the following fields of the `shmid_ds` structure associated with **`shmid`** with the corresponding fields of the structure pointed to by **`buff`**:
	`shm_perm.uid`
	`shm_perm.gid`
	`shm_perm.mode`
	The permission mode occupies nine low-order bits.

`IPC_RMID`	Remove the shared memory segment specified by **`shmid`**.
`SHM_LOCK`	Lock the shared memory segment specified by **`shmid`** in memory.
`SHM_UNLOCK`	Unlock the shared memory segment specified by **`shmid`**.

The `IPC_SET` and `IPC_RMID` commands can only be executed by a process with an effective user ID equal either to super-user or to the value of `shm_perm.cuid` or `shm_perm.uid` in the `shmid_ds` structure associated with **shmid**.

The `SHM_LOCK` and `SHM_UNLOCK` commands can only be executed by a process with an effective user ID equal to super-user.

The `shmid_ds` structure is defined in **shm.h** as follows:

```
struct shmid_ds {
  struct ipc_perm shm_perm;  /* access permissions */
  int shm_segz;    /* segment size in bytes */
  char *shm_reg;    /* not used */
  char pad[4];    /* padding for compatibility */
  unsigned short shm_lpid;  /* last access id */
  unsigned short shm_cpid;  /* creator pid */
  unsigned short shm_nattch;  /* # currently attached*/
  unsigned short shm_cnattch;  /* not used */
  unsigned long shm_atime;  /* time of last attach */
  unsigned long shm_dtime;  /* time of last detach */
  unsigned long shm_ctime;  /* time of creation */
};
```

DIAGNOSTICS

`shmctl` returns 0 if successful, or -1 if not.

If `shmctl` fails, **errno** will contain one of the following:

EINVAL	**shmid** is not a valid shared memory segment identifier.
EINVAL	**cmd** is not a valid command.
EACCES	**cmd** is equal to `IPC_STAT` and read permission is denied to the calling process.
EPERM	**cmd** is equal to `IPC_RMID` or `IPC_SET` and the effective user ID of the calling process is neither super-user nor equal to `shm_perm.cuid` or `shm_perm.uid` in the `shmid_ds` structure associated with **shmid**.
EPERM	**cmd** is equal to `SHM_LOCK` or `SHM_UNLOCK` and the effective user ID of the calling process is not super-user.

EFAULT	**buff** points to an illegal address.
ENOMEM	**cmd** is equal to SHM_LOCK and there is not enough memory.

NAME

shmget – get shared memory segment

SYNOPSIS

```
#include <ipc.h>
#include <shm.h>

int shmget(key, size, flag)
int key;
int size;
int flag;
```

DESCRIPTION

shmget is System V-compatible.

shmget returns the shared memory segment identifier associated with **key**. If the shared memory segment exists, its identifier is returned; if not, a new shared memory segment is created as long as the IPC_CREAT bit is set in **flag**.

A new shared memory segment will always be created if **key** is equal to IPC_PRIVATE.

When a new shared memory segment is created, its associated shmid_ds data structure is initialized as follows:

shm_perm.cuid	set equal to the effective user ID.
shm_perm.uid	set equal to the effective user ID.
shm_perm.cgid	set equal to the effective group ID.
shm_perm.gid	set equal to the effective group ID.
shm_perm.mode	low-order 9 bits set equal to low-order 9 bits of **flag**.
shm_segsz	set equal to **size**.
shm_lpid	set equal to zero.
shm_nattch	set equal to zero.

`shm_atime`	set equal to zero.
`shm_dtime`	set equal to zero.
`shm_ctime`	set equal to the current time.

DIAGNOSTICS

`shmget` returns the shared memory segment identifier if successful, or `-1` if not.

If `shmget` fails, **`errno`** will contain one of the following:

`EINVAL`	**`size`** is either less than or equal to zero or greater than the system limit.
`EINVAL`	A shared memory segment exists for **`key`**, but its size is less than **`size`**.
`EACCES`	A shared memory segment exists for **`key`**, but operation permission is denied according to the low-order 9 bits of **`flag`**.
`ENOENT`	No shared memory segment exists for **`key`** and the `IPC_CREAT` bit is not set in **`flag`**.
`ENOSPC`	A shared memory segment is to be created, but there are no more shared memory segment identifiers.
`ENOMEM`	A shared memory segment is to be created, but there is not enough available memory.
`EEXIST`	A shared memory segment exists for **`key`**, but the `IPC_CREAT` and `IPC_EXCL` bits are both set in **`flag`**.

NAME

shmmap – map a shared memory special file (POSIX.4, Draft 9)

SYNOPSIS

```
#include <sys/shmmap.h>

void *shmmap(fd, addr, len, off, flags)
int fd;
void *addr;
size_t len;
off_t off;
int flags;
```

DESCRIPTION

shmmap is used to map all or part of the shared memory identified by the open file descriptor **fd** to the process' address space. Upon successful completion, the region of the file consisting of **len** contiguous bytes starting from the offset specified by **off** will be mapped into the process' address space. If the address is used by the process, then that address is not an appropriate address at which to establish mapping.

The **addr** argument must be a multiple of 4K so that it falls on a page boundary.

The **offset** argument must be a multiple of 4K so that it falls on a page boundary.

If the system cannot determine an appropriate address, the shmmap call will fail. The **flags** argument is constructed from the bitwise inclusive OR of zero or more of the following **flags**.

The **flags** are defined in the header **sys/shmmap.h**.

SHM_EXACTexact mapping required
SHM_READrequest read access
SHM_WRITErequest write access
SHM_EXECrequest execute access

Different combinations of the **addr** argument and the SHM_EXACT flag provides for 4 different types of mapping which the user may request:

1. **addr** is NULL, SHM_EXACT not set: The system will determine an appropriate address at which to map the file region.
2. **addr** is NULL, SHM_EXACT is set: The system will map the file at the address known as the default exact mapping for the file region. If there is no default exact mapping region, the system will map the file at an appropriate address. That address will become the default exact mapping address for future calls to map the same file region. The default exact mapping will remain in effect until all such mappings have been unmapped. Lynx specific: note that once any portion of the file has been mapped, the default exact mapping is set for the whole file.
3. **addr** is non-NULL, SHM_EXACT is set: The file region will be mapped at the address specified by **addr**. If the **addr** argument does not specify an appropriate address at which to establish mapping, the shmmap function will fail.
4. **addr** is non-NULL, SHM_EXACT not set: The file region will be mapped at the requested address if possible, otherwise the system will determine an appropriate address at which to map the file region.

The file region will be mapped with the access permissions specified by the SHM_READ, SHM_WRITE, and SHM_EXEC **flags**. If these **flags** request more access than allowed by the file mode bits (as modified by the oflag argument of open), the shmmap will fail. For SHM_EXEC to succeed, the file must have execute permission and must have been opened with read permission.

While a mapping is in effect, a subsequent close of the file descriptor **fd** will not affect the mapping or the contents of the shared memory special file. The mapping can be unmapped by calling shmunmap, the process exiting or the process calling exec. Any mappings made by a process are inherited by the child on fork.

DIAGNOSTICS

If successful, `shmmap` returns a pointer to the address space at which the file region is mapped. If unsuccessful, `shmmap` returns NULL and sets **`errno`** to indicate the error as follows.

EBADF	The **`fd`** argument is not a valid open file descriptor.
EBUSY	The SHM_EXACT bit is set in **`flags`** and **`addr`** specifies a mapping that cannot be established.
EINVAL	The value of **`flags`**, **`offset`**, or **`len`** is not valid.
ENOMEM	There is no address space available to satisfy the requested mapping.

CONFORMANCE

`shmmap` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `open(), close()`

The calls that operate on POSIX shared memory special files include:

System Calls - `mkshm(), shmmap(), shmunmap(), open(), close(), unlink()`

Example - `shmmap.c`

NAME

`shmop: shmat, shmdt` – shared memory operations

SYNOPSIS

```
#include <ipc.h>
#include <shm.h>

char *shmat(shmid, shmaddr, flag)
int shmid;
char *shmaddr;
int flag;

int shmdt(shmaddr)
char *shmaddr;
```

DESCRIPTION

`shmat` is System V-compatible.

`shmat` attaches the shared memory segment associated with identifier **`shmid`** to the data segment of the calling process. The address at which the segment is attached depends on the value of **`shmaddr`**:

- If **`shmaddr`** is zero, the segment is attached at an address selected by the operating system.
- If **`shmaddr`** is non-zero and the `SHM_RND` bit is set in **`flag`**, the segment is attached at the address given by the highest multiple of `SHMLBA` less than or equal to **`shmaddr`**.
- If **`shmaddr`** is non-zero and the `SHM_RND` bit is not set in **`flag`**, the segment is attached at the address given by **`shmaddr`**.

The segment is attached for reading if the `SHM_RDONLY` bit is set in **`flag`**; otherwise it is attached for reading and writing.

`shmdt` detaches the shared memory segment located at address **`shmaddr`** from the data segment of the calling process.

DIAGNOSTICS

`shmat` returns the address of the attached shared memory segment if successful, or -1 if not.

If `shmat` fails, **errno** will contain one of the following:

EINVAL	**shmid** is not a valid shared memory segment identifier.
EINVAL	**shmaddr** is non-zero and the highest multiple of SHMLBA that is less than or equal to **shmaddr** is an illegal address.
EINVAL	**shmaddr** is non-zero, the SHM_RND bit is not set in **flag**, and **shmaddr** is an illegal address.
EACCES	operation permission is denied to the calling process.
ENOMEM	The size of the shared memory segment exceeds the process' available data space.
EMFILE	The number of shared memory segments attached to the calling process would exceed the system limit.

`shmdt` returns 0 if successful, or -1 if not.

If `shmdt` fails, **errno** will contain the following:

EINVAL	**shmaddr** is not the address of a shared memory segment.

NAME

shm_open – open and initialize a shared memory object

SYNOPSIS

```
#include <sys/mman.h>

int shm_open(name, oflags, mode)
const char *name;
int oflags;
mode_t mode;
```

DESCRIPTION

shm_open initializes and opens a shared memory object with the name specified by the character string **name**, depending on the argument **oflags**. It returns an open file descriptor associated with the shared memory object. This file descriptor can then be used by other functions (such as mmap) to refer to that shared memory object. The FD_CLOEXEC flag is set for the new file descriptor. This means that upon invocation of an exec family function, the file descriptor is closed. **name** must be constructed according to the standard pathname construction rules, even though it is not in the file system.

The **oflags** argument determines whether the shared memory is opened or initialized. The **oflags** argument may be determined from the bitwise inclusive OR of the following flags, whichever are defined in the header <fcntl.h>. Exactly one of the following two flags must be specified in the **oflags** argument:

O_RDONLY Open for read only access

O_RDWR Open for read and write access

Any combination of the following flags may be specified in the **oflags** argument:

O_CREAT If the shared memory object does not exist, it is created. Upon creation, the shared memory object has a size of zero. The **mode** argument determines the permission

	bits of the shared memory object. If the shared memory object exists, this flag has no effect, except as noted under `O_EXCL` below.
`O_EXCL`	If the `O_EXCL` and `O_CREAT` flags are set together, `shm_open` fails if the shared memory object exists.
`O_TRUNC`	If the shared memory object exists, and is opened with `O_RDWR`, the object is truncated to zero length.

DIAGNOSTICS

`shm_open` initializes and opens a shared memory object when successful and returns `the` open file descriptor. On failure it returns `-1`, and sets **`errno`** as follows:

`EACCES`	The calling process does not have permission to create or open the shared memory object.
`EEXIST`	The shared memory object already exists, and `O_CREAT` and `O_EXCL` flags were specified.
`EINTR`	A signal interrupted the `shm_open` call.
`EMFILE`	Too many file descriptors are being used by the calling process.
`ENAMETOOLONG`	The **`name`** argument is too long, or a specific pathname component is too long.
`ENFILE`	Too may shared memory objects are currently open in the system.
`ENOENT`	The specified shared memory object does not exist, and the `O_CREAT` flag is not set.
`ENOSPC`	There is not sufficient space for the creation of the new shared memory object.

CONFORMANCE

`shm_open` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `mmap(), ftruncate(), close()`

NAME

`shm_unlink` – remove a shared memory object

SYNOPSIS

```
int shm_unlink(name)
const char *name;
```

DESCRIPTION

`shm_unlink` removes the shared memory object named by the string **`name`**. If there are one or more references to the shared memory object when the object is unlinked, only the name is removed when `shm_unlink` returns. The memory object contents are removed only when all open and map references to the shared memory object cease to exist.

DIAGNOSTICS

`shm_unlink` removes the shared memory object when successful, and returns `0`. Otherwise it returns `-1`, and sets **`errno`** to one of the following values:

`EACCES`	The calling process does not have permission to unlink the named shared memory object.
`ENAMETOOLONG`	The length of the name of the shared memory object is too long, or the specific pathname component is too long.
`ENOENT`	The named shared memory object does not exist.

CONFORMANCE

`shm_unlink` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls - `shm_open()`

NAME

`shmunmap` – unmap a shared memory special file (POSIX.4, Draft 9)

SYNOPSIS

```
#include <sys/shmmap.h>

int shmunmap(addr, len)
void *addr;
sizet len;
```

DESCRIPTION

`shmunmap` is used to unmap the shared memory region from the process' address space. If **addr** is the address returned from a previous call to `shmmap`, and **len** is 0, then on successful completion of the `shmunmap` call, the shared memory special file region previously mapped at **addr** is unmapped. In addition, any locks placed on the region by `memlock` will be unlocked.

If the `shmunmap` function removes the last mappings on the shared memory special file region, the open count of the shared memory special file is zero, and there are no links to the shared memory special file, then the shared memory special file is removed. If the `shmunmap` function removes the last mapping on a non-persistent shared memory special file, and the file is not an open file, then the contents of the file will become undefined.

DIAGNOSTICS

If successful, `shmunmap` returns zero. If unsuccessful, `shmunmap` returns -1 and sets **errno** to indicate the error as follows.

EINVAL	The **addr** argument does not describe a currently mapped shared memory special file region which can be unmapped; **len** is not zero.

CONFORMANCE

`shmunmap` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

System Calls - `open(), close(), unlink()`

The calls that operate on POSIX shared memory special files include:

System Calls - `mkshm(), shmmap(), shmunmap(), open(), close(), unlink()`

Example - `shmunmap.c`

NAME

`shutdown` – shut down part of a full-duplex connection

SYNOPSIS

```
shutdown(s, how)
int s, how;
```

DESCRIPTION

`shutdown` causes all or part of a full-duplex connection on the socket associated with **s** to be shut down. If **how** is `0`, then further receives will be disallowed. If **how** is `1`, then further sends will be disallowed. If **how** is `2`, then further sends and receives will be disallowed.

DIAGNOSTICS

`shutdown` returns `0` if successful, or `-1` if not.

If the call fails, **errno** will contain one of the following:

EBADF	**s** is not a valid descriptor.
ENOTSOCK	**s** is a file, not a socket.
ENOTCONN	The specified socket is not connected.

NOTE

`shutdown` has local significance only; no information is propagated to the remote host.

SEE ALSO

System Calls - `connect(), socket()`

NAME

`sigaction` – examine or specify the action associated with a signal

SYNOPSIS

```
#include <signal.h>

int sigaction(sig, act, oact)
int sig;
struct sigaction *act, *oact;
```

DESCRIPTION

`sigaction` provides a POSIX-compliant interface to the signal management facilities. It is functionally identical to `sigvec` except that `sigaction` accepts a 64-bit signal mask for masking out POSIX.4 real-time extended signals (signals numbered 33 through 64) as well as signals numbered 1 through 32.

A signal is an asynchronous or synchronous software event, conceptually similar to a hardware interrupt or exception. When a signal is received, the context of the receiving thread is saved and the handler for the signal is executed. When the handler is invoked, the signal is added to the signal mask and thus blocked. When the handler returns, the old context (including the old signal mask) is restored. Synchronous signals are similar to hardware exceptions, and in many cases are caused by exceptions. The synchronous signals are `SIGILL`, `SIGTRAP`, `SIGFPE`, `SIGBUS`, and `SIGSEGV`. When one of these signals is generated by an exception, the signal is sent to the offending thread.

Any signal, synchronous or not, can be generated manually by a user program using the kill system call. The target can be either a thread or a process. If a signal is directed at a thread, then the signal will be handled by that thread, unless the thread is currently performing a `sigwait` for that signal, in which case the thread's `sigwait` will return and the signal will be discarded. If a signal is directed at a process, the procedure is slightly different: If any thread within the process is currently performing a `sigwait` for that signal, then one such thread will be notified by having its `sigwait` return, and the signal will be discarded. Otherwise, the signal will be handled by the initial thread of the process.

A thread may arrange for any function or subroutine to be used as the handler for a signal. If no handler is assigned, a default action is taken, which may be to suspend the receiving process, to kill the thread or process, or to do nothing. All handlers are per-thread; that is, if one thread sets up a handler for a particular signal, it does not affect the handler for any other thread's signals. A thread may also ignore a signal or block a signal, independently of other threads. An ignored signal is discarded if received; a blocked signal remains pending until it is unblocked.

Any signal in the range 33 through 64 is known as a real-time extended signal. A real-time extended signal differs from other signals in that more than one real-time signal of a particular number can be pending at once. For example, a thread can have 2 (or more) pending real-time signal 45s, but no more than one pending signal 11. Also, a real-time signal can be sent accompanied by a 4-byte data value, whereas a signal cannot.

`sigaction` examines and/or specifies the action associated with signal **`sig`**. A signal action is specified by a `sigaction` structure, defined in **`signal.h`** as follows:

```
struct sigaction {
    void        (*sa_handler)();
    sigset_t    sa_mask;
    int         sa_flags;
};
```

The `sa_handler` field identifies the action to be associated with the specified signal. If `sa_handler` is a function address, then `sa_mask` is added to the set of blocked signals for the thread before the function is invoked. The signal **`sig`** is automatically added to this set. When the signal handler returns, the original set of blocked signals is restored.

The `sa_flags` field is used to modify the actions that take place when the signal occurs, and should be either zero or the bitwise-OR of one or more of the following values (defined in **`signal.h`**):

`SA_NOCLDSTOP` If **`sig`** is `SIGCHLD`, then the system will not generate a `SIGCHLD` signal for the calling process whenever any of its child processes stop.

`SA_ABORT`	Lynx is a pre-emptable real-time multi-tasking OS. This flag is ignored, but is maintained for purposes of backwards compatibility. It has no special effect.
`SA_SYSVCONTEXT`	This flag is ignored, but is maintained for purposes of backwards compatibility. It has no special effect.
`SA_SIGINFO`	This flag must be set for a real-time extended signal in order for proper POSIX.4 Draft 14 signal delivery to occur. This is the standard POSIX.4 signal delivery mechanism as defined in POSIX.4.
`SA_D9EV`	This flag, Lynx-proprietary, is provided for backwards compatibility. If it is set for a real-time signal, then the POSIX.4 Draft 9 signal delivery mechanism is used for deliveries of this signal. Support for this flag may be discontinued by Lynx in the future.

If neither `SA_SIGINFO` nor `SA_D9EV` are set for a real-time extended signal, then the delivery mechanism consists of a handler called with two arguments: signal number and data value. This corresponds to the delivery mechanism provided by POSIX.4 Draft 10. It is recommended to use `SA_SIGINFO`, the standard, portable solution. Both `SA_D9EV` and this default format are subject to eventual obsolescence.

Any use of `SA_SIGINFO` in your process will assure that all real-time extended signals have a default action of process termination; any use of `SA_D9EV`, conversely, will set the default signal action for this process to discard the signal. Otherwise, the default signal action for the real-time signals is a configurable option: either to discard the signal (as in Draft 9 of POSIX.4) or terminate the process (as inPOSIX.4 Draft 14). As shipped, the default behavior is that of Draft 14: process termination. See the kernel configuration guide for more details.

`sigaction` associates with signal **sig** the action specified in the structure pointed to by **act** if **act** is not NULL. If **act** is NULL, then the action associated with signal **sig** is unchanged. In either case, the previous action associated with signal **sig** is returned in the structure pointed to by **oact**, if **oact** is not NULL.

There are three types of actions that can be associated with a signal, as specified in `act.sa_handler`:

`SIG_DFL` Perform the system default action. Refer to the sigvec manual page for a listing of the default actions for all signals. If a default action specifies that a "core dump" be performed, then a core file will be placed in the current working directory of the process as long as the following are true:

The effective user ID and the real user ID of the process are equal.

A file named "core" either doesn't already exist or is writable. If the core file is created, its owner and group IDs will be the same as the effective user and effective group IDs of the process, respectively, and its mode will be 0666 modified by the process' file creation mode mask.

If the default action is to ignore the signal, then any pending signal **sig** is discarded, whether or not it is blocked.

`SIG_IGN` Ignore the signal. This causes any pending signal **sig** to be discarded and any future occurrences of signal **sig** to be ignored. In addition, if **sig** is `SIGCHLD`, then child processes will not become zombies when they terminate.

function address

Upon receipt of the signal, call the function pointed to by `act.sa_handler`. This function is known as the signal handler. In general, the process will resume execution at the point where it was interrupted by the signal after the signal handler function has returned. `sigaction` does not ensure that `act.sa_handler` is a valid function address. If `act.sa_handler` is invalid, then the results are undefined when the signal **`sig`** occurs.

A handler routine can be declared:

```
void sigxhandler(signumber, code, context, fpcontext)
int signumber;
int code;
struct sigcontext *context;
struct fp_context *fpcontext;
```

The arguments `signumber` and `code` identify the exception; `signumber` contains the signal number and `code` contains signal-dependent auxiliary information (such as a floating point error `code`). The `sigcontext` structure (defined in **`signal.h`**) in `context` contains (among other things) the program counter at the point the signal was received. By modifying this `context`, the thread may influence its flow of control after the signal has been handled. If the signal was generated due to a floating point exception, then `fpcontext` refers to the floating point `context` at the time of the exception; this state may be used to determine the reason for the floating point exception. Neither `context` nor `fpcontext` are passed to the handlers for signals unless `SA_SYSVCONTEXT` was specified in the most recent call to `sigaction` for this particular signal.

There are two ways of defining the handler routine. One for non real-time signals (1-32) and the other for the real-time signals (33-64). A handler for non-real-time signals can be declared as follows:

```
int signal_handler ( int sig_num )
{
}
```

For real-time signals two extra fields are defined:

```
int signal_handler ( int sig_num, siginfo_t *extra,
   void *v )
{
}
```

where `siginfo_t` is defined as:

```
typedef struct siginfo {
    int si_signo;
    union sigval    si_value;
    int si_code;
} siginfo_t;
```

For draft 14 real time queued signals, `sigval` is defined as:

```
union sigval {
    int sival_int; /* Integer interpretation. */
    void *sival_ptr; /* Pointer interpretation. */
};
```

Note the important difference between ignoring a signal and blocking it. If a signal is sent but blocked by the target thread, it is not discarded and will take effect as soon as it is removed from the signal mask. If a signal is sent but ignored by the target thread, it is discarded; no record of the signal having been sent is kept. Even if the signal is later given a handler or reset to the default action, the discarded signal is lost.

The signal mask is inherited from the process' parent.

`sigaction` always cancels a pending signal **sig**, except for a pending `SIGKILL` signal.

There are 64 possible signals, numbered 1 through 64. The first 32 are non real-time signals and the second 32 are real-time signals. Most of the signals have a system definition, but some are entirely user-defined. Refer to the `sigvec` manual page for a listing of all signals.

If any stop signal (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`) is delivered to a thread, any pending `SIGCONT` signal is discarded. Conversely, if `SIGCONT` is delivered to a thread, all pending stop signals are discarded. If `SIGCONT` is delivered to a stopped thread, the thread is resumed even if the thread has blocked or ignored `SIGCONT`. If `SIGCONT` is blocked and not ignored, it remains pending until it is either unblocked or a stop signal is delivered to the thread.

The signals SIGKILL and SIGSTOP cannot be blocked, ignored, or caught, and the signal SIGPRIO cannot be blocked or caught. Any attempt to violate these restrictions will cause sigaction to fail with a return value of -1 and **errno** set to EINVAL.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using the signal SIGTHREADKILL. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle SIGTHREADKILL.

DIAGNOSTICS

sigaction returns 0 if the signal action was set, or -1 if not.

If the call fails, **errno** will contain one of the following:

EFAULT	**act** or **oact** does not point to allocated memory in the process' address space.
EINVAL	**sig** is an invalid signal number or equal to SIGKILL or SIGSTOP.
EINVAL	**sig** is equal to SIGPRIO and act.sa_handler is not equal to SIG_DFL or SIG_IGN.

SEE ALSO

System Calls - execve(), fork(), kill(), sigblock(), sigpause(), sigpending(), sigprocmask(), sigsetmask(), sigsuspend(), sigvec(), sigwait()

NAME

`sigblock` – block signals

SYNOPSIS

```
#include <signal.h>

long sigblock(mask)
long mask;
```

DESCRIPTION

`sigblock` sigblock adds the set of signals in mask to the set of signals that are blocked for the calling thread. If signal **s** is to be blocked, bit **s-1** in **mask** should be set. A code segment to block the signal `SIGINT` might look like this:

```
oldmask = sigblock(1L << (SIGINT - 1));
```

When a signal is received by a thread whose signal mask is such that the signal is blocked (the corresponding bit is `1`), no immediate action is taken. Should the thread later change the signal mask so that the signal is unblocked, the action appropriate to the signal will immediately be taken.

Blocking a signal is done on a per-thread basis, just as handling a signal and ignoring a signal are. In other words, if one thread blocks a particular signal, it does not affect the blockage of any other thread's signals.

If a stop signal (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`) is delivered to a thread, any pending `SIGCONT` signal is discarded. Conversely, if `SIGCONT` is delivered to a thread, all pending stop signals are discarded. If `SIGCONT` is delivered to a stopped thread, the thread is resumed even if the thread has blocked or ignored `SIGCONT`. If `SIGCONT` is blocked and not ignored, it remains pending until it is either unblocked or a stop signal is delivered to the thread.

Blocking a signal will not prevent `sigwait` from intercepting delivery of that signal. In fact, blocking all signals and then waiting for the signals with `sigwait` is the appropriate way to handle signals in a multithreaded process.

The signals `SIGKILL`, `SIGSTOP`, and `SIGPRIO` cannot be blocked. An attempt to block these signals will fail without a reported error.

The previous set of blocked signals is returned.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using the signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

DIAGNOSTICS

`sigblock` is always successful and returns the previous set of blocked signals.

SEE ALSO

System Calls - `kill()`, `sigvec()`, `sigsetmask()`, `sigwait()`

NAME

`sigpause` – wait for a signal

SYNOPSIS

```
#include <signal.h>

long sigpause(mask)
long mask;
```

DESCRIPTION

`sigpause` sets the signal mask for the calling thread/process to **`mask`** and waits for a signal. No signal will be acknowledged by this thread between the time `sigpause` is called and **`mask`** is installed.

When an unmasked signal does arrive, any appropriate action for that signal is taken. If the action is to terminate the thread or process, then `sigpause` never returns. If the action is to ignore the signal, then `sigpause` continues to wait for other signals. If the action is to invoke a signal handler, then upon return from the signal handler, `sigpause` returns `-1` with **`errno`** set to `EINTR`. The mask previously in effect is restored when `sigpause` returns.

`sigpause` is not affected by signals sent to other threads within the same process. In particular, if a non-initial thread calls `sigpause()`, it will not be awakened by a signal directed at the process, because such a signal is either intercepted by a thread performing `sigwait` or delivered to the initial thread of the process. In general, `sigpause` should not be used in multithreaded programs. Use `sigwait` instead.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

SEE ALSO

System Calls - `kill()`, `sigvec()`, `sigsetmask()`, `sigwait()`

NAME

sigpending – show pending blocked signals

SYNOPSIS

```
#include <signal.h>

int sigpending(set)
sigset_t *set;
```

DESCRIPTION

sigpending returns, in the area pointed to by **set**, the set of signals that are pending for and blocked by the calling process. A signal **s** is in the set if bit **s-1** in **set** is equal to 1.

DIAGNOSTICS

sigpending returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain the following:

EFAULT	**set** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - kill(), sigblock(), sigvec()

NAME

`sigprocmask` – examine and/or change the signal mask for a thread/process

SYNOPSIS

```
#include <signal.h>

long sigprocmask(how, set, oset)
int how;
sigset_t *set, *oset;
```

DESCRIPTION

`sigprocmask` changes the signal mask for the current thread and/or process according to the set of signals pointed to by **set**, if **set** is not NULL, and returns the previous mask in the area pointed to by **oset**, if **oset** is not NULL. The argument **how** indicates the manner in which the signal mask is changed, and should be one of the following values (defined in **signal.h**):

SIG_BLOCK	The resulting mask is the union of the current mask and the set of signals pointed to by **set**.
SIG_UNBLOCK	The resulting mask is the intersection of the current mask and the complement of the set of signals pointed to by **set**.
SIG_SETMASK	The resulting mask is the set of signals pointed to by **set**.

If **set** is NULL, the value of the argument **how** is not significant and the signal mask is unchanged by this call; thus `sigprocmask` can be used to enquire about currently-blocked signals.

When a signal is received by a thread or process whose signal mask is such that the signal is blocked, no immediate action is taken. Should the thread or process later change the signal mask so that the signal is unblocked, the action appropriate to the signal will immediately be taken.

Blocking a signal (i.e. signal masking) is done on a per-thread basis, just as handling a signal and ignoring a signal are. In other words, if one thread blocks a particular signal, it does not affect the blockage of any other thread's signals.

If any stop signal (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`) is delivered to a thread, any pending `SIGCONT` signal is discarded. Conversely, if `SIGCONT` is delivered to a thread, all pending stop signals are discarded. If `SIGCONT` is delivered to a stopped thread, the thread is resumed even if the thread has blocked or ignored `SIGCONT`. If `SIGCONT` is blocked and not ignored, it remains pending until it is either unblocked or a stop signal is delivered to the thread.

Blocking a signal will not prevent `sigwait` from intercepting delivery of that signal. In fact, blocking all signals and then waiting for the signals with `sigwait` is the appropriate way to handle signals in a multithreaded process.

The signals `SIGKILL`, `SIGSTOP`, `SIGCONT`, and `SIGPRIO` cannot be blocked. An attempt to block these signals will fail without a reported error.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

DIAGNOSTICS

`sigprocmask` returns `0` if successful, or `-1` if not.

If the call fails, **`errno`** will contain the following:

`EINVAL`	**`how`** is not one of `SIG_BLOCK`, `SIG_UNBLOCK`, or `SIG_SETMASK`.

SEE ALSO

System Calls - `kill()`, `sigaction()`, `sigblock()`, `sigsetmask()`, `sigvec()`, `sigwait()`

NAME

sigqueue – queue a real-time signal to a process

SYNOPSIS

```
#include <signal.h>

int sigqueue(pid, signo, value)
pid_t pid;
int signo;
const union sigval value;
```

DESCRIPTION

`sigqueue` sends the signal specified by **signo**, along with the value specified by **value** to the process specified by **pid**. If **signo** is zero, no signal is actually sent. This can be used to verify the existence of a valid pid value.

Signals numbered `SIGRTMIN` through `SIGRTMAX` (as defined in **<signal.h>**), are defined as real-time signals. When a signal is generated by the `sigqueue` function, the signal is marked pending. A signal handler for a signal generated by the `sigqueue` function is set with the `sigaction` function. The signal action varies depending on whether the `SA_SIGINFO` bit is set for the signal in the `sa_flags` member of the `sigaction` structure used in `sigaction`.

If the `SA_SIGINFO` flag is not set for the signal, the signal handler function is set with the `sa_handler` member of the `sigaction` structure. This function is entered as:

```
void function(int signo);
```

If the `SA_SIGINFO` flag is set for the signal, the signal handler function is set with the `sa_sigaction` member of the `sigaction` structure. The function is entered as:

```
void function(int signo, siginfo_t *info, void *context);
```

signo is the signal number for the signal being delivered. `info` is a pointer to a `siginfo_t` structure defined in `<signal.h>`. The `context` argument is not used by LynxOS. The `siginfo_t` structure is defined as:

```
int           si_signo     # Signal number
int           si_code      # Cause of the signal
union sigval  si_value     # Signal value
```

The `si_signo` member contains the signal number, and is same as the **signo** parameter. The `si_code` member contains a code for the cause of the signal. The following codes are defined:

SI_USER	Due to a `kill` function.
SI_QUEUE	Due to a `sigqueue` function.
SI_TIMER	Due to the expiration of a timer.
SI_ASYNCIO	Due to the completion of an asynchronous I/O request.
SI_MESGQ	Due to the arrival of a message on an empty message queue.

If `si_code` is one of `SI_QUEUE`, `SI_TIMER`, `SI_ASYNCIO`, or `SI_MESGQ`, the `si_value` member contains an application defined signal value. The `sigval` union is defined in **`<signal.h>`** as:

```
int       sival_int     # Integer signal value
void *    sival_ptr     # Pointer signal value
```

If the application defined value is an integer, the `sival_int` member is used. If it is a pointer, the `sival_ptr` member is used.

If the `SA_SIGINFO` bit is set for a signal generated by the `sigqueue` function, and the signal number is in the range `SIGRTMIN` to `SIGRTMAX`, that signal is queued to the process with the signal value defined by the argument value. Multiple occurrences of signals so generated are queued in FIFO order. Signals generated by the `kill` function or other events (such as alarm timer expiration, bus error, etc.) have no effect on signals already queued for the same signal number.

When multiple, unblocked signals, in the range `SIGRTMIN` to `SIGRTMAX` are pending, the unblocked signal with the lowest signal number in that range is delivered first.

DIAGNOSTICS

`sigqueue` queues the specified signal to the specified process when successful, and returns `0`. On failure it returns `-1` and sets **`errno`** as follows:

`EAGAIN`	No resources available to queue the signal.
`EINVAL`	The **`signo`** argument has an invalid value.
`EPERM`	The calling process does not have proper permission to send the signal to process **`pid`**.
`ESRCH`	The process specified by the **`pid`** argument does not exist.

CONFORMANCE

`sigqueue` is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `kill()`, `sigaction()`, `sigtimedwait()`, `sigwaitinfo()`

NAME

`sigsetmask` – set the signal mask for a process

SYNOPSIS

```
#include <signal.h>

long sigsetmask(mask)
long mask;
```

DESCRIPTION

`sigsetmask` sets the signal mask for the current thread and/or process to **mask**. If signal **s** is to be blocked, bit **s-1** in **mask** should be set.

When a signal is received by a thread whose signal mask is such that the signal is blocked (the corresponding bit is 1), no immediate action is taken. Should the process later, change the signal mask so that the signal is unblocked, the action appropriate to the signal will immediately be taken.

Blocking a signal is done on a per-thread basis, just as handling a signal and ignoring a signal are. In other words, if one thread blocks a particular signal, it does not affect the blockage of any other thread's signals.

If any stop signal (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`) is delivered to a thread, any pending `SIGCONT` signal is discarded. Conversely, if `SIGCONT` is delivered to a thread, all pending stop signals are discarded. If `SIGCONT` is delivered to a stopped thread, the thread is resumed even if the thread has blocked or ignored `SIGCONT`. If `SIGCONT` is blocked and not ignored, it remains pending until it is either unblocked or a stop signal is delivered to the thread.

Blocking a signal will not prevent sigwait from intercepting delivery of that signal. In fact, blocking all signals and then waiting for the signals with sigwait is the appropriate way to handle signals in a multithreaded process.

The signals `SIGKILL`, `SIGSTOP`, and `SIGCONT` cannot be blocked. An attempt to block these signals will fail without a reported error.

The previous set of blocked signals is returned.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread-specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

DIAGNOSTICS

`sigsetmask` is always successful and returns the previous set of blocked signals.

SEE ALSO

System Calls - `kill()`, `sigvec()`, `sigblock()`, `sigwait()`

NAME

sigsuspend – suspend thread/process until signal delivery

SYNOPSIS

```
#include <signal.h>

long sigsuspend(sigmask)
sigset_t *sigmask;
```

DESCRIPTION

sigsuspend is a POSIX-compliant version of sigpause that is functionally identical to sigpause except that sigsuspend accepts a 64-bit signal mask for masking out events (signals numbered 33 through 64) as well as signals numbered 1 through 32.

sigsuspend sets the signal mask for the calling thread/process to the set of signals pointed to by **sigmask** and then suspends the thread/process until delivery of a signal whose action is either to execute a signal handler or to terminate the thread/process. No signal will be acknowledged by this thread/process between the time sigsuspend is called and the new signal mask is installed.

When an unmasked signal does arrive, any appropriate action for that signal is taken. If the action is to terminate the thread/process, the sigsuspend call will not return. If the action is to execute a signal handler, then sigsuspend will return after the handler returns, with the signal mask restored to the set that existed prior to the sigsuspend call.

sigsuspend is not affected by signals sent to other threads within the same process. In particular, if a non-initial thread calls sigsuspend(), it will not be awakened by a signal directed at the process, because such a signal is either intercepted by a thread performing sigwait or delivered to the initial thread of the process. In general, sigsuspend should not be used in multithreaded programs. Use sigwait instead.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using a thread- specific signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

DIAGNOSTICS

There is no successful return value from `sigsuspend` because it suspends the thread indefinitely. On failure, `sigsuspend` returns `-1` and sets **`errno`** to one of the following:

`EFAULT`	`sigmask` does not point to allocated memory in the process' address space.
`EINTR`	A signal was handled by the thread and control has been returned from the handler.
`ENOMEM`	At least one event signal (numbered 33-64) was set in `sigmask`, and the system ran out of memory trying to allocate an internal event queue structure.

SEE ALSO

System Calls - `kill()`, `sigaction()`, `sigpause()`, `sigprocmask()`, `sigsetmask()`, `sigvec()`, `sigwait()`

NAME

sigvec – software exception handling

SYNOPSIS

```
#include <signal.h>

sigvec(sig, vec, oldvec)
int sig;
struct sigvec *vec, *oldvec;
```

DESCRIPTION

A signal is an asynchronous or synchronous software event, conceptually similar to a hardware interrupt or exception. When a signal is received, the context of the receiving thread is saved and the handler for the signal is executed. When the handler is invoked, the signal is added to the signal mask and thus blocked. When the handler returns, the old context (including the old signal mask) is restored.

Synchronous signals are similar to hardware exceptions, and in many cases are caused by exceptions. The synchronous signals are SIGILL, SIGTRAP, SIGFPE, SIGBUS, and SIGSEGV. When one of these signals is generated by an exception, the signal is sent to the offending thread.

Any signal, synchronous or not, can be generated manually by a user program using the kill system call. The target can be either a thread or a process. If a signal is directed at a thread, then the signal will be handled by that thread, unless the thread is currently performing a sigwait for that signal, in which case the thread's sigwait will return and the signal will be discarded. If a signal is directed at a process, the procedure is slightly different: If any thread within the process is currently performing a sigwait for that signal, then one such thread will be notified by having its sigwait return, and the signal will be discarded. Otherwise, the signal will be handled by the initial thread of the process.

A thread may arrange for any function or subroutine to be used as the handler for a signal. If no handler is assigned, a default action is taken, which may be to suspend the receiving process, to kill the thread or process, or to do nothing. All handlers are per-thread; that is, if one thread sets up a handler for a particular signal, it does not affect the handler for any other thread's signals. A thread may also ignore a signal or block a signal, independently of other threads. An ignored signal is discarded if received; a blocked signal remains pending until it is unblocked.

The system call `sigvec` is used to assign a handler for a signal. The structure `sigvec` is defined (in **`signal.h`**) as follows:

```
struct sigvec {
  void (*sv_handler)();
  long sv_mask;
  int sv_onstack;
};
```

The field `sv_handler` points to a routine which will be called when the signal is received. While the handler is running `sv_mask` is added to the set of blocked signals for the thread. The signal being handled is automatically added to this set.

The previous signal vector is written into the structure pointed to by **`oldvec`**, if **`oldvec`** is not NULL.

A handler routine can be declared:

```
void sigxhandler(signumber, code, context, fpcontext)
int signumber;
int code;
struct sigcontext context;
struct fp_context *fpcontext;
```

The arguments `signumber` and `code` identify the exception; `signumber` contains the signal number and `code` contains signal-dependent auxiliary information (such as a floating point error code). The `sigcontext` structure (defined in **`signal.h`**) in `context` contains (among other things) the program counter at the point the signal was received. By modifying this context, the thread may influence its flow of control after the signal has been handled. If the signal was generated due to a floating point exception, then `fpcontext` refers to the floating point context at the time of the exception; this state may be used to determine the reason for the floating point exception.

Neither `context` nor `fpcontext` are passed to the handlers for signals unless `SV_SYSVCONTEXT` was specified in the most recent call to `sigvec` for this particular signal.

The field `sv_onstack` is used to modify the actions that take place when the signal occurs, and should be either zero or the bitwise OR of one or more of the following values (defined in **signal.h**):

SV_ABORT	If the signal is received while in a system call, then the system call is aborted rather than continued. This mimics the behavior of signals (with respect to system calls) under System V UNIX.
SV_FUNCDFL	If `vec.sv_handler` is the address of a routine, then when the signal occurs, the action for the signal will be set to `SIG_DFL` (the signal's default action) before entering the handler, and the action will not be restored after return from the handler. This mimics the behavior of signals under System V UNIX.
SV_SYSVCONTEXT	If `vec.sv_handler` is the address of a routine, then when the signal occurs, the handler will be called with additional arguments representing the `sigcontext` structure in the same format that is used under System V UNIX. In addition, if the signal is `SIGFPE`, then an argument representing a pointer to the `fp_context` structure will follow the `sigcontext` arguments. After return from the handler, the hardware registers will be loaded with the values of the `sigcontext` arguments. This allows a signal handler to modify the values of the hardware registers upon its return.

If `vec.sv_handler` is set to `SIG_DFL` the default action will be taken when the signal is received. If `vec.sv_handler` is set to `SIG_IGN` the signal is ignored.

Note the important difference between ignoring a signal and blocking it. If a signal is sent but blocked by the target process, it is not discarded and will take effect as soon as it is removed from the signal mask. If a signal is sent but ignored by the target process, it is discarded; no record of the signal having been sent is kept. Even if the signal is later given a handler or reset to the default action, the discarded signal is lost.

If a signal is delivered during a system call, the system call continues from where it left off after any signal processing is complete. This is usually the desired behavior, but it is also possible to have the system call abort rather than continue by setting the `SV_ABORT` flag in `vec.sv_onstack`.

The signal mask is inherited from the process' parent. This mask is normally 0 to indicate that no signals are blocked.

There are 32 possible signals, numbered 1 through 32. Most of the signals have a system definition, but some are entirely user-defined. The default action for each signal is described in the table "Default Actions for Signals".

Table 1: Default Actions for Signals

Signal Name	Meaning	Default Action
`SIGHUP`	Hangup	Terminate
`SIGINT`	Interrupt	Terminate
`SIGQUIT`	Quit	Terminate, core dump
`SIGILL`	Illegal instruction	Terminate, core dump
`SIGTRAP`	Trace trap	Terminate, core dump
`SIGBRK`	Break	Terminate, core dump
`SIGCORE`	User defined	Terminate, core dump
`SIGFPE`	Floating point exception	Terminate, core dump
`SIGKILL`	Kill process	Terminate
`SIGBUS`	Bus error	Terminate, core dump
`SIGSEGV`	Segment violation	Terminate, core dump
`SIGSYS`	Bad system call argument	Terminate, core dump
`SIGPIPE`	Write on pipe with no reader	Terminate
`SIGALRM`	Real-time alarm	Terminate
`SIGTERM`	Software termination	Terminate
`SIGURG`	Urgent condition on socket	Ignore

Table 1: Default Actions for Signals

Signal Name	Meaning	Default Action
SIGSTOP	Stop process	Stop
SIGTSTP	Stop process (from tty)	Stop
SIGCONT	Continue after stop	Ignore
SIGCHLD	Child status changed	Ignore
SIGTTIN	Background read from tty	Stop
SIGTTOU	Background write from tty	Stop
SIGIO	I/O possible on descriptor	Ignore
SIGTHREADKILL	Thread cancel	Terminate Thread
SIGFSZ	User defined	Ignore
SIGVTALRM	Virtual timer alarm	Terminate
SIGPROF	Profiling alarm	Terminate
SIGWINCH	Window size change	Ignore
SIGLOST	User defined	Ignore
SIGUSR1	User defined	Terminate
SIGUSR2	User defined	Terminate
SIGPRIO	Priority or process group change	Send FIOPRIO ioctl to all devices

When a thread overruns its stack by no more than one page, the following occurs. First, the stack is extended by one page. Then, a SIGSEGV is sent to the thread. When the thread is done handling the segmentation violation, the newly-allocated page is deallocated again and the thread is sent along its merry way. This behavior does not nest; the thread only has one extra stack page for segmentation violations. If the thread causes stack overruns beyond the extra page while the extra page is still mapped in, then the process is killed with SIGKILL.

Note that a thread can ignore or block SIGSEGV, thus winning itself an extra page of stack. However, this is a stupid thing to do because it is a once-only behavior until the signal is handled, and also because the thread could simply be created with the extra page of stack.

If any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is delivered to a thread, any pending SIGCONT signal is discarded. Conversely, if SIGCONT is delivered to a thread, all pending stop signals are discarded. If SIGCONT is delivered to a stopped thread, the thread is resumed even if the thread has blocked or ignored SIGCONT. If SIGCONT is blocked and not ignored, it remains pending until it is either unblocked or a stop signal is delivered to the thread.

The signals SIGKILL and SIGSTOP cannot be blocked, ignored, or caught, and the signal SIGPRIO cannot be blocked or caught. Any attempt to violate these restrictions will cause sigvec to fail with a return value of **-1** and **errno** set to EINVAL.

NOTE

Thread cancellation and cleanup is implemented outside of the kernel using the signal SIGTHREADKILL. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle SIGTHREADKILL.

DIAGNOSTICS

sigvec returns 0 if the signal vector was set, or -1 if not.

If the call fails, **errno** will contain one of the following:

EFAULT	**vec** or **oldvec** does not point to allocated memory in the process' address space.
EINVAL	**sig** is an invalid signal number or equal to SIGKILL or SIGSTOP.
EINVAL	**sig** is equal to SIGCONT and vec.sv_handler is equal to SIG_IGN.
EINVAL	**sig** is equal to SIGPRIO and vec.sv_handler is not equal to SIG_DFL or SIG_IGN.

SEE ALSO

System Calls - `kill()`, `sigblock()`, `sigsetmask()`, `sigpause()`, `sigwait()`, `execve()`, `fork()`

NAME

`sigwait` – wait for one of a set of signals

SYNOPSIS

```
#include <signal.h>

long sigwait(set, value)
sigset_t *set;
void **value;
```

DESCRIPTION

sigwait waits for one of the signals in the set of signals pointed to by **`set`** to be delivered to the thread. When an awaited signal arrives at the thread or process, the thread is awakened and the number of the arriving signal is returned. If two or more threads are waiting for a given signal, and that signal is sent to the process, then only one of the threads will be awakened with the signal returned to them.

If one of the awaited signals is an event signal (i.e., numbered between 33 and 64, inclusive), and that event arrives causing the `sigwait` to return, then the data value associated with the event will be stored in the location pointed to by value, if value is not NULL. If value is NULL, the event's data value is discarded.

`sigwait` has priority over signal handlers, signal masks, and signal ignoring. This applies to the receiving thread's own signal handlers, masks, and ignoring as well as to the initial threads's signal handlers, masks, and ignoring in the case of a signal directed at the process.

`sigwait` is similar to `sigpause`, but since `sigpause` does not guarantee that the pausing thread will receive a signal if the signal is directed at the process, `sigwait` is the preferred method for dealing with signals in a multithreaded program.

NOTES

Thread cancellation and cleanup is implemented outside of the kernel using the signal, `SIGTHREADKILL`. Programs that want to use thread cancellation and cleanup features should be careful not to block, ignore, unblock, or otherwise handle `SIGTHREADKILL`.

The `sigwait` function description is different between POSIX 1003.4 Draft 9 and POSIX 1003.4 Draft 12. We have implemented `sigwait` to contain the maximum functionality. To achieve the POSIX.4 Draft 9 functionality, the **`value`** argument should be NULL.

The `sigwait` system call will intercept event signals only if they are sent by `kill` and directed at the calling thread. It will not intercept event signals sent by `ekill`, and it will not intercept event signals directed at the process, regardless of how they are sent. Event signals should be caught with `evtpoll`.

DIAGNOSTICS

If `sigwait` succeeds, the number of the received signal is returned. If `sigwait` fails, `-1` is returned.

If the call fails, **`errno`** will contain one of the following:

`EINVAL`	**`set`** contains unstoppable signals.
`EFAULT`	**`set`** does not point to allocated memory in the process' address space.
`EFAULT`	**`value`** is not NULL and does not point to allocated memory in the process' address space.
`EAGAIN`	The calling thread is already doing a `sigwait`.

CONFORMANCE

`sigwait` is implemented according to Draft 4 of the POSIX 1003.4a (Threads) standard.

SEE ALSO

System Calls -	`kill(), sigvec(), sigsetmask(), sigblock(), evtpoll(), ekill()`
Library Functions -	`sigaddset(), sigdelset(), sigemptyset(), sigfillset(), st_cancel(), st_setcancel()`
Example -	`sigwait.c`

NAME

`sigwaitinfo`,
`sigtimedwait` – wait for queued signals

SYNOPSIS

```
#include <signal.h>

int sigwaitinfo(set, info)
const sigset_t *set;
siginfo_t *info;

int sigtimedwait(set, info, timeout)
const sigset_t *set;
siginfo_t *info;
const struct timespec *timeout;
```

DESCRIPTION

`sigwaitinfo` and `sigtimedwait` suspend the calling process and wait for any signal from a specified set of signals.

`sigwaitinfo` waits for a pending signal from the set referenced by the **set** argument. If any multiple signals in the real-time signal range (`SIGRTMIN` to `SIGRTMAX`) are pending, the lowest numbered one is selected. If no signal in **set** is pending, then the calling process is suspended until one or more signals in **set** become pending, or until the process is interrupted by an unblocked, caught signal.

The **info** argument is a pointer to a `siginfo_t` structure, which is defined in **<signal.h>** as

```
int           si_signo     # Signal number
int           si_code      # Cause of the signal
union sigval  si_value     # Signal value
```

If the **info** argument is not `NULL`, the selected signal number is stored in `info->si_signo`. The cause of the signal is stored in `info->si_code`, with the following codes defined:

SI_USER	Due to a kill function.
SI_QUEUE	Due to a sigqueue function.
SI_TIMER	Due to the expiration of a timer.
SI_ASYNCIO	Due to the completion of an asynchronous I/O request.
SI_MESGQ	Due to the arrival of a message on an empty message queue.

Any value queued for the selected signal is dequeued. If more than one values are queued for the same signal, the first queued value is dequeued and stored in info->si_value, if **info** is not NULL.

The sigtimedwait function behaves the same as sigwaitinfo, except that it takes a timeout argument which is a pointer to a **timespec** structure. If no signal in **set** is pending, the calling process is suspended for a time interval specified by the **timespec** structure referenced by **timeout**. If the **timeout** argument points to a zero value interval, and if none of the signals from **set** are pending, sigtimedwait returns immediately with an error.

sigwaitinfo or sigtimedwait may be interrupted by an unblocked signal if that signal is not in **set**.

DIAGNOSTICS

sigwaitinfo and sigtimedwait are successful if one of the signals specified by set is pending or is generated. When successful, both of these functions return the selected signal number. On failure, both of these functions return -1 and set **errno** as follows:

EINTR	The wait was interrupted by a caught signal.
EAGAIN	No signal was delivered within the specified **timeout** range for sigtimedwait.
EINVAL	The **timeout** argument for sigtimedwait contains an invalid tv_nsec field (less than zero or greater than 1000 million).

CONFORMANCE

`sigwaitinfo` and `sigtimedwait` are implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `sigqueue(), sigaction(), sigsuspend()`

NAME

smem_get, smem_create – get a shared memory segment

SYNOPSIS

```
#include <smem.h>

char *smem_get(name, size, perm)
char *name;
long size;
int perm;

char *smem_create(name, physaddr, size, perm)
char *name;
char *physaddr;
long size;
int perm;
```

DESCRIPTION

smem_get makes the shared memory segment named **name** accessible to the calling process. If the named segment does not exist already, and the SM_NCREAT flag is not set in **perm**, then the segment is created. The size of the segment is given (in bytes) by **size**, and is rounded up to a multiple of the system page size. Requesting the wrong size of an existing memory segment is an error.

The call returns a pointer to the base of the shared memory segment. Every task that wishes to share the memory must call smem_get with the same segment **name**. The address returned may be different for different processes; in other words, the segment may be mapped into different portions of each process' virtual address space. When a process calls fork, the child process inherits all shared memory segments referenced by the parent. The virtual address for each shared memory segment is, of course, the same for the parent and child. When a process calls execve, it gives up its shared memory.

Access permissions for shared memory segments are defined by flags set in **perm**: SM_READ indicates read permission, SM_WRITE indicates write permission, and the bitwise OR of both values indicates read and write permission. A fault is generated if the process writes (reads) to the shared memory and does not have write (read) permission.

Shared memory is not "owned" by any process. A shared memory segment will continue to exist even if no process has it mapped to its virtual address space. smem_remove must be called to get rid of a shared memory segment.

smem_create works just like smem_get, but it allows the user to request specific physical memory be used. The base address of the physical memory is given by **physaddr**. This is useful to map special system memory areas (such as a graphics frame buffer) into the address space of a user process.

An important restriction of smem_create is that the requested physical address must be at a 4K page boundary (i.e., it must be evenly divisible by 4096). If the actual address does not meet this restriction, a shared memory region whose base address begins at the nearest (smaller) page boundary address can be created. The requested **size** should take into account the gap from the page boundary to the desired address. The hardware can then be accessed as an offset from the base of the virtual address.

smem_create is also used to "detach" a shared memory segment from a process' address space. The virtual address of the segment should be given in **physaddr** and the SM_DETACH flag should be set in **perm**. The **name** and **size** arguments are ignored. If **physaddr** does not refer to a shared memory segment, then smem_create simply returns without an error. A shared memory segment must be detached by all processes that are using it before it can be removed (see smem_remove).

For VME bus systems, an additional "permission" flag is present in **perm**, SM_SHORTIO. This flag enables a process to access memory or devices which use VME short addressing. (This flag is only meaningful with smem_create.)

DIAGNOSTICS

smem_get returns the address of the shared memory segment if successful, or NULL if not.

If smem_get fails, **errno** will contain one of the following:

EFAULT	**name** does not point to allocated memory in the process' address space.
EINVAL	A shared memory segment was found with a name of **name** but its size does not match **size**.
ENOMEM	The system does not have enough free memory to grant the request.
ENOENT	The shared memory segment is to be created but the SM_NCREAT flag was set in **perm**.

smem_create with the SM_DETACH flag set in **perm** always returns NULL and does not set **errno**. Smem_create without the SM_DETACH flag set returns the address of the shared memory segment if successful, or NULL if not.

If smem_create fails, **errno** will contain one of the following:

EFAUL	**name** does not point to allocated memory in the process' address space.
EINVAL	A shared memory segment was found with a name of **name**, but either its size does not match **size** or it is not physical memory (i.e. not created using smem_create).
EINVAL	The SM_SHORTIO flag was set in **perm** but the value of **physaddr** - DRAMBASE is greater than 64K. On the Lynx30 hardware, DRAMBASE = 0x40000000 (defined in headerslynx30/absolute.h). On all other systems, DRAMBASE = 0 (not defined).
ENOMEM	The system does not have enough free memory to grant the request.

NOTE

smem_create is a license to crash the operating system, and so should be used with caution.

SEE ALSO

System Calls - `smem_remove()`

NAME

smem_remove – remove a shared memory segment

SYNOPSIS

```
smem_remove(name)
char *name;
```

DESCRIPTION

smem_remove removes a shared memory segment from the operating system table of shared memory segments and frees the memory. A shared memory segment must be detached (using smem_create) by all processes that are using it before it can be removed.

DIAGNOSTICS

smem_remove returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain one of the following:

EFAULT	**name** does not point to allocated memory in the process' address space.
EBUSY	Some process has the shared memory segment mapped in its virtual address space.
EINVAL	The shared memory segment does not exist.

SEE ALSO

System Calls - smem_get()

NAME

`socket` – create an endpoint for communication

SYNOPSIS

```
#include <types.h>
#include <socket.h>

s = socket(af, type, protocol)
int s, af, type, protocol;
```

DESCRIPTION

`socket` creates an endpoint for communication and returns a descriptor.

The **af** parameter specifies an address format with which addresses specified in later operations using the `socket` should be interpreted. These formats are defined in the include file **socket.h**. The currently understood format is

AF_INET	(ARPA Internet addresses)
AF_RAWETH	(Ethernet link level addresses)
AF_UNIX	(UNIX internal protocols)

The `socket` has the indicated type which specifies the semantics of communication. Currently defined types are:

```
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_RAWETH
```

A `SOCK_STREAM` `socket` provides a sequenced, reliable, two-way connection based on a byte stream, with an out-of-band data transmission mechanism available. A `SOCK_DGRAM` `socket` supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). `SOCK_RAW` sockets provide access to internal network interfaces. The types `SOCK_RAW` (available only to the super-user), `SOCK_RAWETH` provides interface to the ethernet network interface.

The **protocol** specifies a particular protocol to be used with the socket. Normally, only a single protocol exists to support a particular socket type using a given address format. However, it is possible that many protocols may exist in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the "communication domain" in which communication is to take place; see services and protocols file formats.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect call. Once connected, data may be transferred using read and write calls or some variant of the send and recv calls. When a session has been completed a close may be performed. Out-of-band data may also be transmitted as described in send and received as described in recv.

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable **errno**. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. five minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in send calls. It is also possible to receive datagrams at such a socket with recv.

An fcntl call can be used to specify a process group to receive a SIGURG signal when the out-of-band data arrives.

The operation of sockets is controlled by socket-level options. These options are defined in the file **socket.h** and explained below. setsockopt and getsockopt are used to set and get options, respectively.

SO_DEBUG turn on recording of debugging information

SO_REUSEADDR	allow local address reuse
SO_LINGER	linger on close if data present
SO_DONTLINGER	do not linger on close

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates the rules used in validating addresses supplied in a bind call should allow reuse of local addresses. SO_LINGER and SO_DONTLINGER control the actions taken when unsent messages are queued on socket and a close is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the setsockopt call when SO_LINGER is requested). If SO_DONTLINGER is specified and a close is issued, the system will process the close in a manner which allows the process to continue as quickly as possible.

SOCK_RAWETH type sockets are used for sending and receiving raw ethernet packets. The write and read system calls are used with the sockets of this type to send and receive ethernet packets. A set of special ioctl calls are used to set up various parameters for this operation. Refer to the raw_ethernet(4) manual page for description of these ioctl calls.

DIAGNOSTICS

socket returns a descriptor referencing the socket if successful, or -1 if not.

If the call fails, **errno** will contain one of the following:

EAFNOSUPPORT	The specified address family is not supported in this version of the system.
ESOCKTNOSUPPORT	The specified socket type is not supported in this address family.
EPROTONOSUPPORT	The specified protocol is not supported.
EMFILE	The per-process descriptor table is full.

`ENOBUFS`	No buffer space is available. The `socket` cannot be created.

NOTE

The use of `keepalives` is a questionable feature for this layer.

SEE ALSO

System Calls - `ioctl()`, `accept()`, `bind()`, `connect()`, `getsockname()`, `getsockopt()`, `listen()`, `recv()`, `send()`, `shutdown()`, `socketpair()`

NAME

socketpair – create a pair of connected sockets

SYNOPSIS

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

#include <sys/socket.h>

socketpair(d, type, protocol, sv)

int d, type, protocol;

int sv[2];
```

DESCRIPTION

socketpair creates an unnamed pair of connected sockets in the specified domain of the specified and using the optionally specified. The descriptors used in referencing the new sockets are returned. The two sockets are indistinguishable. A 0 is returned if the call succeeds, -1 if it fails. The call succeeds unless:

[EMFILE] Too many descriptors are in use by this process.

[EAFNOSUPPORT] The specified address family is not supported on this machine.

[EPROTONOSUPPORT]
The specified protocol is not supported on this machine.

[EOPNOSUPPORT] The specified protocol does not support creation of socket pairs.

[EFAULT] The address _s_v does not specify a valid part of the process address space.

NOTE

This call is currently implemented only for the UNIX domain.

SEE ALSO

System Calls - `read(), write(), pipe()`

NAME

`stat, lstat, fstat` – get the status of a file

SYNOPSIS

```
#include <types.h>
#include <stat.h>

int stat(path, status)
char *path;
struct stat *status;

int lstat(path, status)
char *path;
struct stat *status;

int fstat(fd, status)
int fd;
struct stat *status;
```

DESCRIPTION

`stat` returns all information stored about a file in the inode of the file. The information is written into a **`stat`** structure pointed to by **`status`**, defined in **`stat.h`** as follows:

```
struct stat {
  dev_t   st_dev;     /* block device inode is on */
  ino_t   st_ino;     /* inode number */
  int     st_mode;    /* protection and file type */
  int     st_nlink;   /* hard link count */
  int     st_uid;     /* user id */
  int     st_gid;     /* group id */
  dev_t   st_rdev;    /* the device number for a special file */
  off_t   st_size;    /* number of bytes in a file */
  time_t  st_atime;   /* time of last access */
  time_t  st_mtime;   /* time of last modify */
  time_t  st_ctime;   /* time of last status change */
  long    st_blksize; /* size of a block */
  long    st_blocks;  /* number of blocks in the file */
};
```

The `st_mode` field contains the permission and file type information, as defined by the following constants (from `stat.h`):

S_IFMT	0170000	Isolates file type bits
S_IFDIR	0040000	Directory
S_IFCHR	0020000	Character special
S_IFBLK	0060000	Block special
S_IFREG	0100000	Regular file
S_IFLNK	0120000	Symbolic link
S_IFSOCK	0140000	Socket
S_ISVTX	0001000	Save text image
S_IREAD	0000400	Read permission
S_IWRITE	0000200	Write permission
S_IEXEC	0000100	Execute permission

The file type can be determined by forming the bitwise-AND of the constant `S_IFMT` with the value of `st_mode`, then comparing this result with one of `S_IFDIR`, `S_IFCHR`, `S_IFBLK`, `S_IFREG`, `S_IFLNK`, or `S_IFSOCK`.

`lstat` works just like **stat**, unless the path refers to a symbolic link, in which case the information about the link is returned. `stat` traverses symbolic links, and returns information about the file referenced.

`fstat` returns the status information for a file, given an open file descriptor instead of a pathname.

DIAGNOSTICS

All of these routines return 0 if successful in retrieving status information, or -1 if not.

If `stat` or `lstat` fails, **errno** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.

`ENOENT`	The file does not exist.
`ELOOP`	Too many symbolic links were encountered when traversing the path.
`EACCES`	A directory in the path denied search permission.
`EFAULT`	**`path`** does not point to allocated memory in the process' address space.
`EFAULT`	**`status`** does not point to allocated memory in the process' address space.

If `fstat` fails, **`errno`** will contain one of the following:

`EBADF`	**`fd`** is not a valid file descriptor.
`EFAULT`	**`status`** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - `chmod()`

NAME

`statfs, fstatfs` – get file system information

SYNOPSIS

```
#include <types.h>
#include <statfs.h>

int statfs(path, buf, len, fstyp)
char *path;
struct statfs *buf;
int len;
int fstyp;

int fstatfs(fd, buf, len, fstyp)
int fd;
struct statfs *buf;
int len;
int fstyp;
```

DESCRIPTION

`statfs` is System V-compatible.

`statfs` returns information about a file system in a file system independent superblock. It can be used to get information about mounted or unmounted file systems. In either case, the information is written into a **statfs** structure pointed to by **buf**, defined in **statfs.h** as follows:

```
struct statfs {
  short f_fstyp;  /* file system type */
  long f_bsize;   /* block size */
  long f_frsize;  /* fragment size (not supported) */
  long f_blocks;  /* # of blocks on this file system */
  long f_bfree;   /* # of free blocks */
  long f_files;   /* # of inodes on this file system */
  long f_ffree;   /* number of free inodes */
  char f_fname[6];  /* volume name */
  char f_fpack[6];  /* pack name */
};
```

statfs fills in this structure up to a maximum of **len** bytes. **len** must be no greater than `sizeof` (`struct statfs`) and will typically be exactly that value.

If the file system is currently mounted, **path** should name a file which resides on that file system. In this case the file system type is known to the operating system and the **fstyp** argument must be zero. If the file system is not mounted, **path** must name the block device special file containing it, and **fstyp** must contain the (non-zero) file system type. In both cases, `read`, `write`, or `execute` permission of the named file is not required, but all directory components of **path** must be searchable.

`fstatfs` performs the same operation as `statfs`, but the file is specified by an open file descriptor instead of by name.

The `statfs` system call obsoletes `ustat` and should be used in preference to it in future programs.

DIAGNOSTICS

Both of these routines return 0 if successful, or -1 if not.

If `statfs` fails, **errno** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
ELOOP	Too many symbolic links were encountered when traversing the path.
EACCES	A directory in the path denied search permission.
EFAULT	**path** does not point to allocated memory in the process' address space.
EFAULT	**buf** does not point to allocated memory in the process' address space.
EINVAL	**fstyp** is an invalid file system type.

EINVAL	**path** is not a block device special file and **fstyp** is non-zero.
EINVAL	**len** is negative or is greater than sizeof (struct statfs)

If fstatfs fails, **errno** will contain one of the following:

EBADF	**fd** is not a valid file descriptor.
EFAULT	**buf** does not point to allocated memory in the process' address space.
EINVAL	**fstyp** is an invalid file system type.
EINVAL	**fd** does not refer to a block device special file and **fstyp** is non-zero.
EINVAL	**len** is negative or is greater than sizeof (struct statfs)

SEE ALSO

System Calls -	chmod(), chown(), link(), mknod(), pipe(), read(), unlink(), write()
File Formats -	inode, superblock

NAME

st_build – create a new system thread

SYNOPSIS

```
#include <time.h>

#include <st.h>

tid_t st_build(start_routine, stksize, prio, arg)
void *(*start_routine)();
int stksize;
int prio;
void *arg;
```

DESCRIPTION

st_build builds a new system thread (**st**) in the current process context. The **st** is initialized to begin running the routine at **start_routine**, which is passed the single parameter **arg**.

stksize determines the **st**'s stack size in bytes. This number is a minimum; the stack of the **st** will probably be rounded up to an integral number of pages.

prio sets the **st**'s scheduling priority. It must be a number between 0 and 255.

The **st** is created in the stopped state and should be started at will with st_resume. The **st**'s initial cancellation state is CANCEL_GENERAL.

DIAGNOSTICS

st_build returns the ID of the new thread when successful, or -1 upon error. If the call fails, **errno** will contain one of the following:

EINVAL	The stack size given is bogus.
EINVAL	The priority value is not a valid priority.
EFAULT	**start_routine** does not point to a location in the process' address space.

`ENOMEM`	Out of room for **`st`**.

CONFORMANCE

None. POSIX threads should be used instead of the system thread facilities.

SEE ALSO

System Calls -	`st_exit()`, `st_join()`, `fork()`
Library Functions -	`st_default_attr()`, `st_setcancel()`, `st_testcancel()`, `st_cancel()`

NAME

st_detach – free resources associated with a system thread

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_detach(tid)
tid_t tid;
```

DESCRIPTION

st_detach indicates to the system that no further **st**'s will be attempting to join with the **st** whose ID is given by **tid**. The resources associated with the **st** can be immediately freed when it st_exits or is st_canceled.

st_detach does not free those resources necessary to run the **st** unless it has already st_exited or been successfully st_canceled. Thus, a **st** can be detached while still running and will continue to execute as before.

Subsequent st_joins with this system thread will return with **errno** set to EINVAL.

DIAGNOSTICS

st_detach returns 0 when successful and -1 when unsuccessful. If the call fails, **errno** will contain the following:

EINVAL	The **st** ID given does not refer to a currently-existing system thread

CONFORMANCE

None. POSIX threads should be used instead of the system thread facilities.

SEE ALSO

System Calls - st_join(), st_exit(), _cancel(), _exit()

NAME

st_exit – terminate the calling system thread

SYNOPSIS

```
#include &lt;time.h&gt;

#include &lt;st.h&gt;

int st_exit(status)
void *status;
```

DESCRIPTION

st_exit terminates the **st** calling it. The value status becomes the **st**'s exit status, and is passed to other **st**'s who are doing a st_join with this **st**. The system thread's exit status remains available to other **st**'s via st_join until a st_detach is performed to free up all resources associated with this **st**.

If any cleanup routines have been pushed onto the thread's cleanup stack and not popped when the thread calls st_exit, then each routine and its argument shall be popped off the thread cleanup stack and called with that argument.

DIAGNOSTICS

st_exit never returns.

CONFORMANCE

None. POSIX threads should be used instead of the system thread facilities.

SEE ALSO

System Calls - st_join(), st_detach(), _exit()

Library Functions - st_cancel(), st_pushrecover(), st_poprecover()

NAME

st_join – await the termination of another system thread.

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_join(tid, status)
tid_t tid;
void **status;
```

DESCRIPTION

st_join blocks the calling **st** until the system thread whose ID is given by **tid** terminates. At that time, the **st** performing the st_join is unblocked and the exit status of the terminated **st** is returned in the status argument.

Behavior With Respect To Thread Cancellation

A call to st_join is a thread cancellation point. That is, if a thread has set general cancelability and not asynchronous cancelability, and if a cancellation request is issued against this thread before or during the duration of the call to st_join, the thread will be canceled.

Special exit statuses are used to indicate that a **st** was st_canceled by another **st** rather than st_exiting normally.

DIAGNOSTICS

st_join returns 0 when successful, and -1 when it fails. When st_join fails, **errno** will be set to one of the following values:

EINVAL	The value specified by tid does not refer to any existing **st**.
ESRCH	The **st** specified by **tid** exists but has been st_detached.

CONFORMANCE

None. POSIX threads should be used instead of the system thread facilities.

SEE ALSO

System Calls - `st_exit()`, `st_detach()`, `st_cancel()`, `_exit()`

NAME

st_prioresume – switch from one thread to another with priority inheritance

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_prioresume(newtid, newprio)
tid_t newtid;
int newprio;
```

DESCRIPTION

st_prioresume sets the priority of the calling thread to **newprio**, and resumes the thread named by **newtid** in the same way as st_resume.

DIAGNOSTICS

st_prioresume returns 0 when successful, or -1 upon error. If the call fails, **errno** will contain one of the following:

ESRCH	The thread named by **newtid** cannot be found.
EINVAL	The priority value is not a valid priority.

SEE ALSO

System Calls - st_stop(), st_resume(), st_switch(), setpriority()

NAME

`st_resume` – resume execution of a system thread

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_resume(tid)
tid_t tid;
```

DESCRIPTION

`st_resume` resumes the execution a **st** previously stopped by `st_stop` or created via `st_new` with the `attr.state` argument set to `THREAD_STOPPED`, or a system thread created using `st_build`. A resumed **st** will not necessarily run immediately; it may be blocked on some other event, like I/O or a `st_join`. However, a stopped system thread is guaranteed not to run, even if it is otherwise runnable.

`st_resume` and `st_stop` maintain a counter of their respective calls. Thus, a system thread that has been `st_resumed` 10 times must be `st_stopped` 10 times before it actually is stopped.

NOTE

An **st** that has called `st_exit` can still be the successful target of an `st_resume`; however, don't expect it to start running—it's finished. The thread ID, however, remains valid until `st_detach` is called.

DIAGNOSTICS

`st_resume` returns 0, or -1 upon error. If an error occurs, **errno** will be set to one of the following values:

`EINVAL`	The value of **tid** does not refer to a currently valid **st**.

SEE ALSO

System Calls - `st_new(), st_stop()`

NAME

st_setabstimer – absolute timer manipulation/blocking routine

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_setabstimer(tid, abstime)
tid_t tid;
struct timeval *abstime;
```

DESCRIPTION

st_setabstimer allows threads to sleep until absolute times (that is, times that are not relative to the current time). Alternatively, st_setabstimer allows threads to manipulate the wakeup times of other threads that have put themselves to sleep using st_setabstimer.

If **tid** is the ID of the calling thread, the thread will sleep until the absolute time specified by **abstime**, unless there is some reason for premature awakening.

If **tid** is the ID of another thread, st_setabstimer will arrange for that thread to be awakened at the absolute time indicated by **abstime**. This awakening assumed that **tid** is already asleep due to a call of its own to st_setabstimer. If the time indicated by **abstime** has already passed, **tid** is immediately awakened.

The time is given as the number of seconds and microseconds since the beginning of LynxOS time, normally midnight GMT on January 1, 1970. The time passed to st_setabstimer should not include any local timezone or daylight savings adjustments.

If **abstime** is NULL, and **tid** is the current thread, st_setabstimer returns successfully immediately. If **abstime** is NULL, and **tid** is another thread, st_setabstimer awakens the other thread immediately.

DIAGNOSTICS

When called with **tid** referring to another thread, `st_setabstimer` returns 0 when successful. When called with **tid** referring to the current thread and after sleeping the entire time indicated by **abstimer** `st_setabstimer` will return -1 and set **errno** equal to `EINTR`. If the thread is awakened prematurely by another thread calling `st_setabstimer`, 0 is returned. In either case, -1 is returned upon error.

If the call fails, **errno** will contain one of the following:

`EINVAL`	The thread ID passed is bogus or the thread is dead.
`EFAULT`	The **timeval** argument is a bad pointer other than NULL.
`EAGAIN`	The thread is not the current thread and the absolute time has passed.

SEE ALSO

System Calls - `gettimeofday()`

NAME

`st_sethandle` – set handle for thread-specific data

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_sethandle(addr)
void **addr;
```

DESCRIPTION

`st_sethandle` set the address of the thread-specific data handle for the calling process. All threads within this process will have distinct contents for this address maintained and switched by the kernel, so that the address becomes, essentially, an extra register for the threads.

NOTE

`st_sethandle` is used to implement thread-specific data. Applications should never call `st_sethandle` directly.

DIAGNOSTICS

`st_sethandle` returns 0 when successful, or -1 upon error. If the call fails, **errno** will contain the following:

EINVAL	The address is bad for any of a number of reasons. Currently, address must be within the test, data, or BSS sections of the process. Thread stack addresses are not yet allowed.

CONFORMANCE

None. POSIX threads should be used instead of the system thread facilities.

SEE ALSO

System Calls - `st_build(), st_resume(), st_stop()`

NAME

`st_stop` – stop a system thread

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_stop(tid)
tid_t tid;
```

DESCRIPTION

`st_stop` stops the **st** identified by **tid**. A stopped **st** will not run until it is started again by a call to `st_resume`. There are only two ways to stop a `st`—either by issuing `st_stop`, or by creating a **st** via `st_new` with the `attr.state` argument set to `THREAD_STOPPED`.

`st_resume` and `st_stop` maintain a counter of their respective calls. Thus, a system thread that has been `st_resumed` 10 times must be `st_stopped` 10 times before it actually is stopped.

NOTE

An **st** that has called `st_exit` can still be the successful target of an `st_stop`; however, don't expect any real effect—the thread has finished. The thread ID, however, remains valid until `st_detach` is called.

DIAGNOSTICS

`st_stop` returns 0, or -1 upon error. If an error occurs, **errno** will be set to the following value:

`EINVAL`	The value of **tid** does not refer to a currently valid **st**.

SEE ALSO

System Calls - `st_new(), st_resume(), st_build()`

NAME

`st_stopself` – stop self with timeout

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_stopself(timeout)
struct timeval *timeout;
```

DESCRIPTION

`st_stopself` is equivalent to a call to `st_stop` with a thread ID argument of 0; that is, `st_stopself` decrements the start count of the calling thread, and stops the thread if the start count becomes negative. If the thread is not awakened within the time interval indicated by **`timeout`**, though, the thread will be awakened after the fashion of `st_resume`.

DIAGNOSTICS

`st_stopself` returns -1 upon error. A successful return is one where another thread awakened the calling thread before the **`timeout`** expired. If the **`timeout`** does expire, -1 will be returned and **`errno`** will be set to `EINTR`. If the call fails, **`errno`** will contain one of the following:

`EINTR`	The requested time passed without any other thread awakening this thread.
`EFAULT`	The time pointer passed in refers to a bad address.
`EAGAIN`	Temporary resource overrun: no timeouts are available.

SEE ALSO

System Calls - `st_stop(), st_resume()`

NAME

st_switch – switch from one thread to another with priority inheritance

SYNOPSIS

```
#include <time.h>

#include <st.h>

int st_switch(newtid, newprio)
tid_t newtid;
int prio;
```

DESCRIPTION

st_switch sets the priority of thread **newtid** to **newprio**, suspends the calling thread in the same way as st_stop, and resumes **newtid** in the same way as st_resume.

DIAGNOSTICS

st_switch returns 0 when successful, or -1 upon error. If the call fails, **errno** will contain one of the following:

EINVAL	The thread ID passed is bogus.
EINVAL	The priority value is not a valid priority.

SEE ALSO

System Calls - st_stop(), st_resume(), st_prioresume(), setpriority()

NAME

`symlink` – make a symbolic link to file

SYNOPSIS

```
symlink(path1, path2)
char *path1;
char *path2;
```

DESCRIPTION

A symbolic link named **path2** is created to the file named **path1**. The file can then be accessed either through **path2** or **path1**. **Path1** and **path2** need not be on the same file system. **path1** may or may not exist; **path2** must not exist.

DIAGNOSTICS

`symlink` returns -1 if the link could not be made, otherwise 0.

If the call fails, **errno** will contain one of the following:

ENOTDIR	An element of either path prefix is not a directory.
ENAMETOOLONG	**path1** or **path2** is too long.
EEXIST	**path2** does exist.
ELOOP	Too many symbolic links were encountered when traversing one of the paths.
EROFS	Making the link requires writing to a directory on a read-only file system.
EACCES	Making the link requires writing to a directory that denies write permission.
EFAULT	**path1** or **path2** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - `link(), readlink()`

NAME

`sync` – update all file systems

SYNOPSIS

```
int sync()
```

DESCRIPTION

`sync` causes the entire disk cache and the modified in-core `inodes` to be written out to disk. The call returns after scheduling the write operations; it does not wait for the completion of the operations.

SEE ALSO

System Calls - `fsync()`

NAME

`sysctl` – get or set system information

SYNOPSIS

#include <sys/sysctl.h>

int sysctl(int ***name*****, u_int** *namelen***, void ****oldp***, size_t ****oldlenp***, void ****newp***, size_t** *newlen***);**

DESCRIPTION

The `sysctl function retrieves system information and allows processes` with appropriate privileges to set system information. The information available from `sysctl` consists of integers, strings, and tables.

Unless explicitly noted below, `sysctl` returns a consistent snapshot of the data requested. Consistency is obtained by locking the destination buffer into memory so that the data may be copied out without blocking. Calls to `sysctl` are serialized to avoid deadlock.

The state is described using a “Management Information Base” (MIB) style name, listed in name, which is a namelen length array of integers.

The information is copied into the buffer specified by *oldp*. The size of the buffer is given by the location specified by *oldlenp* before the call, and that location gives the amount of data copied after a successful call. If the amount of data available is greater than the size of the buffer supplied, the call supplies as much data as fits in the buffer provided and returns with the error code `ENOMEM`. If the *old* value is not desired, *oldp* and *oldlenp should be set to* `NULL`.

The size of the available data can be determined by calling `sysctl` with a NULL parameter for *oldp*. The size of the available data will be returned in the location pointed to by *oldlenp*. For some operations, the amount of space may change often. For these operations, the system attempts to round up so that the returned size is large enough for a call to return the data shortly thereafter.

To set a new value, *newp* is set to point to a buffer of length *newlen* from which the requested value is to be taken. If a new value is not to be set, *newp* should be set to `NULL` and *newlen* set to 0.

The top level names are defined with a `CTL_ prefix` in `<sys/sysctl.h>`.

CTL_NET

The string and integer information available for the CTL_NET level is detailed below. The changeable column shows whether a process with appropriate privilege may change the value.

Second level name	Type	Changeable
PF_ROUTE	routing messages	no
PF_INET	internet values	yes

`PF_ROUTE` Return the entire routing table or a subset of it. The data is returned as a sequence of routing messages (see `route`(4) for the header file, format and meaning). The length of each message is contained in the message header.

`PF_INET` Get or set various global information about the internet protocols. The third level name is the protocol. The fourth level name is the variable name. The currently defined protocols and names are:

Protocol name	Variable name	Type	Changeable
ip	forwarding	integer	yes
ip	redirect	integer	yes
ip	ttl	integer	yes
icmp	maskrepl	integer	yes
udp	checksum	integer	yes

The third level name is a protocol number, which is currently always 0. The fourth level name is an address family, which may be set to 0 to select all address families. The fifth and sixth level names are as follows:

Fifth level name	Sixth level is:
NET_RT_FLAGS	rtflags
NET_RT_DUMP	None
NET_RT_IFLIST	None

The variables are as follows:

`ip.forwarding`	Returns 1 when IP forwarding is enabled for the host, meaning that the host is acting as a router.
`ip.forwarding`	Returns 1 when ICMP redirects may be sent by the host. This option is ignored unless the host is routing IP packets, and should normally be enabled on all systems.
`ip.forwarding`	The maximum time-to-live (hop count) value for an IP packet sourced by the system. This value applies to normal transport protocols, not to ICMP.
`ip.forwarding`	Returns 1 if ICMP network mask requests are to be answered.
`udp.checksum`	Returns 1 when UDP checksums are being computed and checked. Disabling UDP checksums is strongly discouraged.

RETURN VALUES

If the call to `sysctl` is successful, 0 is returned. Otherwise –1 is returned and errno is set appropriately.

ERRORS

The following errors may be reported:

`[EFAULT]`	The buffer *name, oldp, newp,* or the length pointer *oldlenp* contains an invalid address.
`[EINVAL]`	The name array is less than two or greater than `CTL_MAXNAME`.
`[EINVAL]`	A non-null *newp* is given and its specified length in newlen is too large or too small.
`[ENOMEM]`	The length pointed to by *oldlenp* is too short to hold the requested value.
`[ENOTDIR]`	The *name* array specifies an intermediate rather than terminal name.

`[EOPNOTSUPP]`	The *name* array specifies a value that is unknown.
`[EPERM]`	An attempt is made to set a read-only value.
`[EPERM]`	A process without appropriate privilege attempts to set a value.

FILES

`<sys/sysctl.h>`	definitions for top level identifiers, second level kernel and hardware identifiers, and user level identifiers
`<sys/socket.h>`	definitions for second level network identifiers
`<netinet/in.h>`	definitions for third level Internet identifiers and fourth level IP identifiers
`<netinet/icmp_var.h>`	definitions for fourth level ICMP identifiers
`<netinet/udp_var.h>`	definitions for fourth level UDP identifiers

HISTORY

The `sysctl` function first appeared in 4.4BSD.

NAME

`sysfs` – get file system type information

SYNOPSIS

```
#include <fstyp.h>
#include <fsid.h>

int sysfs(opcode, fsname)
int opcode;
char *fsname;

int sysfs(opcode, fsindex, buf)
int opcode;
int fsindex;
char *buf;

int sysfs(opcode)
int opcode;
```

DESCRIPTION

`sysfs` is System V-compatible.

`sysfs` returns information about the file system types configured in the system. The number of arguments accepted by `sysfs` varies and depends on the value of **`opcode`**. The available queries are given by the following values for **`opcode`**:

`GETFSIND` Return the file-system type index associated with the given null-terminated file-system identifier, **`fsname`**.

`GETFSTYP` Return, in the buffer pointed to by **`buf`**, the file-system type identifier associated with the given file-system type index, `fs_index`. The buffer pointed to by **`buf`** must be at least of size `FSTYPSZ` as defined in **`fstyp.h`**.

`GETNFSTYP` Return the total number of file system types configured in the system.

LynxOS currently supports only its own file system, so that the only valid file-system index is 1, the only valid file-system identifier is “LYNX”, and the number of file system types configured in the system is always 1.

DIAGNOSTICS

`sysfs` returns a non-negative integer if successful, or -1 if not.

If `sysfs` fails, **errno** will contain one of the following:

EINVAL	**fsname** points to an invalid file-system identifier.
EINVAL	**fs_index** is zero, or invalid.
EINVAL	**opcode** is not one of GETFSIND, GETFSTYP, or GETNFSTYP.
EFAULT	**buf** does not point to allocated memory in the process’ address space.
EFAULT	**fsname** does not point to allocated memory in the process’ address space.

NAME

`sysi86` – perform Intel x86 system-specific operation

SYNOPSIS

```
#include <fp.h>

sysi86(cmd, arg)
int cmd;
char *arg;
```

DESCRIPTION

`sysi86` provides a way of accomplishing Intel x86-based system specific operations.

The operation code is given in **cmd** as follows:

`SI86FPHW` Provide floating point coprocessor information in **arg**. The value returned in **arg** is:

`FCOP_NO`	No floating point coprocessor installed.
`FCOP_287`	80287 floating point coprocessor installed.
`FCOP_387`	80387 floating point coprocessor installed.

DIAGNOSTICS

`sysi86` returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain the following:

`EINVAL` Request or **arg** is invalid.

AVAILABILITY

Intel x86 platforms only.

NAME

`sysMC30` – perform Motorola 68030-based system-specific operation

SYNOPSIS

```
#include <fp.h>

sysMC30(cmd, arg)
int cmd;
char *arg;
```

DESCRIPTION

`sysMC30` provides a way of accomplishing Motorola 68030-based system specific operations.

The operation is given in **cmd** as follows:

`SMC30FPHW` Provide floating point coprocessor information in **arg**. The value returned in **arg** is:

- `FP_NO` No floating point coprocessor installed.
- `FP_81` MC68881 floating point coprocessor installed.
- `FP_82` MC68882 floating point coprocessor installed.

DIAGNOSTICS

`sysMC30` returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain the following:

`EINVAL` Request or **arg** is invalid.

NAME

`sysMC40` – perform Motorola 68040-based system-specific operation

SYNOPSIS

```
#include <sysMC40.h>

sysMC40(cmd, arg)
int cmd;
int *arg;
```

DESCRIPTION

`sysMC40` provides a way of accomplishing Motorola 68040-based system specific operations.

The operation is given in **`cmd`** as follows:

`SYSCONFIG` Provide system configuration information in **`arg`**. The value returned in **`arg`** is bitwise `OR` of the following bits (as applicable to the underlying hardware):

`IS_33MHZ`	CPU clock speed is 33MHz; otherwise it is 25MHz.
`IS_VMEBUS`	VME chip installed on-board for VMEbus access.
`IS_SCSI`	SCSI chip installed on-board.
`IS_LAN`	LAN chip installed on-board.
`IS_68040`	Processor type is 68040; otherwise it is 68LC040.

DIAGNOSTICS

`sysMC40` returns `0` if successful, or `-1` if not.

If the call fails, **`errno`** will contain the following:

`EINVAL` Request or **`arg`** is invalid.

NAME

`sysppc` – enable/disable floating point exceptions (PowerPC)

SYNOPSIS

```
#include <fp.h>

sysppc ( command, argument )
int command;
int argument;

sysppc(command)
int command;
```

DESCRIPTION

The `sysppc` system call is used to enable/disable floating point exceptions. Also, `sysppc` can be used to set a floating point exception type for such instances as: dividing by zero, invalid operation exceptions, and overflow or underflow exceptions.

The **command** argument can take the following values:

- `FP_EXC_SIGNALLING`
- `GET_FPSCR_REG`
- `SET_FPSCR_REG`
- `GET_MSR_REG`

If **command** is `FP_EXC_SIGNALLING`, it can take the arguments `FPE_ENABLE` or `FPE_DISABLE`. If **command** is `SET_FPSCR_REG`, it can take the logical OR values of `FP_VE`, `FP_OE`, `FP_UE`, `FP_ZE`, or `FP_XX`. These definitions are found in `<fp.h>`.

EXAMPLES

```
sysppc(FP_EXC_SIGNALLING, FPE_ENABLE);
sysppc(FP_EXC_SIGNALLING, FPE_DISABLE);
sysppc(SET_FPSCR_REG, FP_VE | FP_OE | FP_UE | FP_ZE);
fpscr = sysppc(GET_FPSCR_REG);
```

NAME

`timer_create` – create a per-process timer

SYNOPSIS

```
#include <time.h>
#include <signal.h>

int timer_create(clock_id, evp, timerid)
clockid_t clock_id;
struct sigevent *evp;
timer_t *timerid;
```

DESCRIPTION

The `timer_create` function creates a per-process timer based on the clock **clock_id**. A timer ID used to identify the timer is returned in the location referenced by **timerid**. The timer is in a disarmed state when `timer_create` returns.

The **evp** argument is used to define the asynchronous notification that will occur when the timer expires. The structure `sigevent` consists of the following members:

```
int           sigvec_signo     # Signal number
union sigval  sigvec_value     # Signal value
int           sigvec_notify    # Notification type
```

A POSIX.1b real-time signal handler should be set if a real-time signal is desired to be queued to the process upon timer expiration. Refer to the `sigqueue` manual page for more information about real-time signal handlers.

If the `sigev_notify` member of **evp** is set to `SIGEV_SIGNAL`, and if the `SA_SIGINFO` flag is set for the expiration signal, the signal and the value defined in `sigev_value` is queued to the process upon timer expiration.

If the **evp** argument is `NULL` and the clock is set to `CLOCK_REALTIME`, then the signal `SIGALRM` is generated upon timer expiration.

When the process forks a child, the child process does not inherit the per-process timers of the parent process. When the process makes an `exec` call, all per-process timers are disarmed and deleted.

DIAGNOSTICS

The `timer_create` function creates a new timer when successful, and returns `0`. Otherwise it returns a value of `-1` indicating failure, and sets **`errno`** to one of the following values:

`EAGAIN`	The calling process has reached the limit on number of per-process timers for this process. There are no sufficient signal queuing resources available.
`EINVAL`	The **`clock_id`** argument does not correspond to any known clock.

CONFORMANCE

The `timer_create` function is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls -	`sigqueue()`, `timer_delete()`, `timer_settime()`, `timer_gettime()`
Example -	`timer_create.c`

NAME

`timer_delete` – delete a per-process timer

SYNOPSIS

```
#include <time.h>

int timer_delete(timerid)
timer_t *timerid;
```

DESCRIPTION

The `timer_delete` function deletes the timer **`timerid`**. If the timer is armed, it is disarmed before the deletion.

DIAGNOSTICS

The `timer_delete` function deletes a timer when successful, and returns `0`. Otherwise it returns a value of `-1` indicating failure, and sets **`errno`** to the following value:

`EINVAL`	The **`timerid`** argument is an invalid timer ID.

CONFORMANCE

The `timer_delete` function is implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

Library Calls -	`timer_create()`, `timer_settime()`, `timer_gettime()`

NAME

`timer_settime`,
`timer_gettime`,
`timer_getoverrun` – per-process timer functions

SYNOPSIS

```
#include <time.h>

int timer_settime(timerid, flags, value, ovalue)
timer_t timerid;
int flags;
const struct itimerspec *value;
struct itimerspec *ovalue;

int timer_gettime(timerid, value)
timer_t timerid;
struct itimerspec *value;

int timer_getoverrun(timerid)
timer_t timerid;
```

DESCRIPTION

The `timer_settime` function sets the expiration time and the timer interval reload value for the timer **timerid**. Both of these values are determined from the **value** argument, which is a structure of type **itimerspec**. The **itimerspec** structure is defined in **<time.h>** as:

```
struct timespec  it_value      # Timer expiration
struct timespec  it_interval   # Timer interval
```

The expiration time of **timerid** is set to the `it_value` member of the **value** argument. If the timer was already armed when `timer_settime` is called, the expiration time for the timer is reset based on the value argument. If the `it_value` member of the **value** argument is zero, the timer is disarmed.

The **flags** argument determines whether the timer expiration is set as an absolute value, or relative to the clock on which the timer is based. If the flag TIMER_ABSTIME is set in the **flags** argument, the timer will expire when the absolute value of the clock reaches the value equal to the it_value member from the **value** argument. If this time has already passed, the timer_settime function succeeds and an expiration notification is made. If the flag TIMER_ABSTIME is not set in the **flags** argument, the timer will expire after it_value nanoseconds from when the call is made.

The reload interval for **timerid** is set in the it_interval member from the **value** argument. If the it_interval member is not zero, a periodic timer is initiated.

If the **ovalue** argument is not NULL, the timer_settime function sets the same values as would be retrieved by a timer_gettime call (described below) into the **ovalue** argument.

The timer_gettime function retrieves the amount of time until the next expiration of the timer **timerid**, and its reload value into the **value** argument. The it_value member of the value argument is set to the amount of time until the timer expiration, or zero if the timer has expired. This value is always returned relative to the next expiration, even if the timer was set with the TIMER_ABSTIME flag. The it_interval member of the **value** argument is set to the reload value last set by a timer_settime call.

The timer_getoverrun function returns the count of expiration overruns for the timer **timerid**. An overrun is defined as the event when a timer for which a signal is still pending expires. For any given timer, only a single signal will be queued to the process at any time. No new signal is queued when an overrun occurs. The value returned by the timer_getoverrun function is equal to the number of extra timer expirations that occurred between the time when a signal was generated and when it was delivered, up to a maximum of DELAYTIMER_MAX. If more than DELAYTIMER_MAX overruns occur for the timer, timer_getoverrun returns DELAYTIMER_MAX.

DIAGNOSTICS

`timer_settime` and `timer_gettime` set a timer and retrieve the value of a timer, respectively, when successful and return a value of `0`. The `timer_getoverrun` function returns the timer expiration overrun count when successful. All of these functions return `-1` on failure, and set **`errno`** as follows:

`EINVAL` The **`timerid`** argument does not correspond to a valid timer.

The **`value`** argument specified to the `timer_settime` function specified a nanosecond value less than zero or greater than or equal to 1000 million.

CONFORMANCE

The `timer_settime`, `timer_gettime`, and `timer_getoverrun` functions are implemented according to the POSIX 1003.1b (Real-Time) standard.

SEE ALSO

System Calls - `timer_create()`, `timer_delete()`

NAME

`times` – get process and child process times

SYNOPSIS

```
#include <types.h>
#include <times.h>

clock_t times(buff)
struct tms *buff;
```

DESCRIPTION

`times` populates the **tms** structure pointed to by **buff** with user and system time consumption information for the calling process and each of its terminated children for which it has executed a `wait` system call. The **tms** structure is defined in **times.h** as follows:

```
struct tms {
  clock_t tms_utime;  /* User time for this process */
  clock_t tms_stime;  /* System time for this process */
  clock_t tms_cutime; /* User time for all children */
  clock_t tms_cstime; /* System time for all children */
};
```

`tms_utime` is the CPU time used by the calling process in user mode.

`tms_stime` is the CPU time used by the calling process in system mode.

`tms_cutime` is the sum of the `tms_utimes` and `tms_cutimes` of the child processes.

`tms_cstime` is the sum of the `tms_stimes` and `tms_cstimes` of the child processes.

All `times` are reported in clock ticks. The number of clock ticks per second is defined by the constant `TICKSPERSEC`, defined to be 100 in `conf.h`.

DIAGNOSTICS

`times` returns the number of clock ticks that have elapsed since system start-up time if successful, or `-1` if not.

If `times` fails, **`errno`** will contain one of the following:

`EFAULT`	**`buff`** does not point to allocated memory in the process' address space.

SEE ALSO

System Calls - `getrusage()`

NAME

`truncate, ftruncate` – reduce size of a file

SYNOPSIS

```
truncate(name, size)
char *name;
long size;

ftruncate(fd, size)
int fd;
long size;
```

DESCRIPTION

`truncate` and `ftruncate` set the size in bytes of a file to **`size`**. `truncate` operates on the file named **`name`**, while `ftruncate` operates on the open file referenced by file descriptor **`fd`**. For `ftruncate`, the file must be open for writing. If the file size is already less than or equal to **`size`**, it remains unchanged. If the file size is greater than **`size`**, it is set to **`size`** and all bytes beyond are released to the system. Bytes in the last block of the file which is beyond the logical end of file are not cleared or overwritten.

DIAGNOSTICS

`truncate` and `ftruncate` return 0 if successful, or -1 if not.

If `truncate` or `ftruncate` fail, **`errno`** will contain one of the following:

`ENOTDIR`	An element of the path prefix is not a directory.
`ENAMETOOLONG`	The path name is too long.
`ENOENT`	The file does not exist.
`ELOOP`	Too many symbolic links were encountered when traversing the path.
`EACCES`	A directory in the path denied search permission.
`EFAULT`	**`name`** does not point to allocated memory in the process' address space.

EROFS	The file is on a read-only file system.
ETXTBSY	The file is an executable program currently in use.
EISDIR	The file is a directory.

`ftruncate` may also fail for the following reasons:

EBADF	**fd** is not a valid file descriptor.
EINVAL	**fd** does not refer to a regular file.
EINVAL	The file referenced by **fd** is not open for writing.
EROFS	The file is on a read-only file system.

SEE ALSO

System Calls - `stat()`, `lseek()`, `unlink()`

NAME

`ulimit` – get and set user limits

SYNOPSIS

```
long ulimit(cmd, newlimit)
int cmd;
long newlimit;
```

DESCRIPTION

`ulimit` is System V-compatible.

`ulimit` provides various controls over process limits. The operations are given by the following values for **`cmd`**:

1 Return the limit, in 512-byte blocks, on the size of a regular file that can be created by the calling process. Currently, this limit is always $(2^{31}-1)/512 = 4194303$ because under LynxOS, the only limit on the size of a file is the limit imposed by the addressing range of a 32-bit signed integer, which is $2^{31}-1$ bytes.

2 Under System V UNIX, this **`cmd`** is used to set the limit, in 512-byte blocks, on the size of a regular file that can be created by the calling process to **`newlimit`**. Since LynxOS does not impose any limit on the size of a file (besides the $2^{31}-1$ addressing range limit), this **`cmd`** has no effect. For compatibility reasons, `ulimit` will fail if a non-superuser process tries to set the limit above 4194303.

3 Return the maximum break value that can be set using the `brk` system call.

4 Return the value of `USR_NFDS`, the maximum number of open files per process.

DIAGNOSTICS

`ulimit` returns a non-negative integer if successful, or -1 if not.

If `ulimit` fails, **`errno`** will contain one of the following:

`EPERM`	The process attempted to set the limit on the size of a regular file above 4194303, and the process is not a super-user process.
`EINVAL`	**`cmd`** is not one of 1, 2, 3, or 4.

SEE ALSO

System Calls - `brk()`

NAME

`umask` – change the file creation mode mask for a process

SYNOPSIS

```
include <stat.h>

int umask(mask)
int mask;
```

DESCRIPTION

`umask` sets the file creation mode mask for the calling process to **`mask`**. The old mask is returned. Bits that are set in the file creation mode mask are cleared in the modes of files created by the process. The mode bits may be the bitwise-OR of any number of the following (defined in **`stat.h`**):

S_IRUSR	read permission: owner
S_IRGRP	read permission: group
S_IROTH	read permission: other
S_IWUSR	write permission: owner
S_IWGRP	write permission: group
S_IWOTH	write permission: other
S_IXUSR	execute permission: owner
S_IXGRP	execute permission: group
S_IXOTH	execute permission: other

SEE ALSO

System Calls - `chmod(), stat(), open()`

NAME

`uname` – get name of current LynxOS system

SYNOPSIS

```
#include <sys/utsname.h>

int uname(name)
struct utsname *name;
```

DESCRIPTION

`uname` populates the **`utsname`** structure pointed to by **`name`** with information about the current LynxOS system. The **`utsname`** structure is defined in **`sys/utsname.h`** as follows:

```
struct utsname {
  char sysname[MAXHOSTNAMELEN+1];
  char nodename[MAXHOSTNAMELEN+1];
  char release[9];
  char version[9];
  char machine[9];
};
```

`sysname` is a null-terminated character string containing the name of the LynxOS system.

`nodename` is a null-terminated character string containing the name that the system is known by on a network.

`release` is a null-terminated character string identifying the release of LynxOS.

`version` is a null-terminated character string identifying the version under this release of LynxOS.

`machine` is a null-terminated character string identifying the hardware on which the LynxOS system is running.

DIAGNOSTICS

`uname` returns 0 if successful, or -1 if not.

If `uname` fails, **`errno`** will contain the following:

EFAULT	**`name`** does not point to allocated memory in the process' address space.

NAME

`unlink` – remove a file entry from a directory

SYNOPSIS

```
int unlink(path)
char *path;
```

DESCRIPTION

`unlink` removes the file entry named **path** from its directory. If the file has no other directory entry and the file is not in use, all resources of the file are returned to the system. If the file is open, the resources are returned when the last process using the file closes it.

`unlink`ing a directory is not allowed. Directories should be removed with `rmdir`.

For POSIX shared memory special files, the action of `unlink` is the same as for regular files except that if one or more references to the file exist when the last link is removed, the link will be removed, but the removal of the file contents will be postponed until all open and map references to the file have been removed.

DIAGNOSTICS

`unlink` returns 0 if successful, or -1 if not.

If the call fails, **errno** will contain one of the following:

ENOTDIR	An element of the path prefix is not a directory.
ENAMETOOLONG	The path name is too long.
ENOENT	The file does not exist.
ELOOP	Too many symbolic links were encountered when traversing the **path**.
EROFS	The file is on a read-only file system.
EACCES	A directory in the path denied search permission.

`EACCES`	The directory the entry is in denies write permission.
`EFAULT`	**`path`** does not point to allocated memory in the process' address space.
`EPERM`	**`path`** refers to a directory.

SEE ALSO

System Calls - `close(), link(), rmdir(), stat()`

Calls that operate on POSIX shared memory special files include:

System Calls - `mkshm(), shmmap(), shmunmap()`

NAME

`ustat` – get file system statisticsf

SYNOPSIS

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

int ustat(dev, buff)
devt dev;
struct ustat *buff;
```

DESCRIPTION

`ustat` is System V-compatible.

`ustat` populates the `ustat` structure pointed to by **buff** with information about the file system mounted on device **dev**. The `ustat` structure is defined in **ustat.h** as follows:

```
struct ustat {
  long           f_tfree;  /* number of free blocks */
  unsigned short f_tinode; /* number of free inodes */
  char           f_fname[6];
  char           f_fpack[6];
};
```

`f_fname` and `f_fpack` will always contain the null string.

DIAGNOSTICS

`ustat` returns 0 if successful, or -1 if not.

If `ustat` fails, **errno** will contain one of the following:

EINVAL	**dev** is not the device number of a device containing a mounted file system.
EFAULT	**buff** does not point to allocated memory in the process' address space.

NAME

`utimes` – set file times

SYNOPSIS

```
#include <time.h>

int utimes(path, tvp)
char *path;
struct timeval *tvp;
```

DESCRIPTION

`utimes` uses the access time contained in **`tvp[0]`** and the modification time contained in **`tvp[1]`** to set the corresponding times for the file whose path name is pointed to by **`path`**. Both times are measured in the number of seconds that have elapsed since midnight GMT, 1 January 1970.

The "inode-changed" time (`ctime`) of the file is set to the current time.

If **`tvp`** is NULL, then no action is performed.

DIAGNOSTICS

`utimes` returns `0` if successful, or `-1` if not.

If `utimes` fails, **`errno`** will contain one of the following:

`ENOTDIR`	An element of the path prefix is not a directory.
`ENAMETOOLONG`	The path name is too long.
`ENOENT`	The file does not exist.
`ELOOP`	Too many symbolic links were encountered when traversing the path.
`EACCES`	A directory in the path denied search permission.
`EROFS`	The file is on a read-only file system.
`EFAULT`	**`tvp`** is not NULL and does not point to allocated memory in the process' address space.

`EFAULT`	**`path`** does not point to allocated memory in the process' address space.

NAME

`vmstart` – activate demand paged virtual memory

SYNOPSIS

```
int vmstart(fd, size)
int fd;
long size;
```

DESCRIPTION

`vmstart` activates demand paged virtual memory to the swap file specified by the open file descriptor **`fd`**. **`size`** gives the size of the swap space in bytes. **`size`** will be rounded down to the nearest page. For maximum efficiency, it is best to use a contiguous file or raw partition as the swap file. `vmstart` can be called only by a process with effective user ID of 0 (super-user).

If the **`fd`** is `-1`, then instead of paging using a swap file, application text is paged directly from the file system (including file systems in ROM) and application data and bss is not swapped. **`size`** should be the amount of virtual memory that the text for all applications will need.

NOTE

`size` must be a positive number and must be larger than physical memory.

DIAGNOSTICS

`vmstart` returns `0` if successful, or `-1` if not.

If `vmstart` fails, **`errno`** will contain one of the following:

`EPERM`	The effective user ID of the calling process is not `0`.
`EBUSY`	Demand paged virtual memory is already active.
`EBADF`	**`fd`** is not a valid file descriptor.
`EINVAL`	**`fd`** is not an open file descriptor.

`EINVAL`	**`size`** is less then size of physical memory.
`EROFS`	**`fd`** refers to a file on a read-only file system.
`ENOMEM`	There is not enough free memory in the system to allocate the virtual memory allocation table.

SEE ALSO

System Calls - `mkcontig()`

NAME

vmtopm – convert virtual addresses to physical addresses

SYNOPSIS

```
#include <mem.h>

int vmtopm(pid, array, base, size)
int pid;
struct dmachain *array;
char *base;
long size;
```

DESCRIPTION

vmtopm **writes** into **array** the physical addresses corresponding to a virtual address range beginning at **base** and **size** bytes long, for the process named by **pid**. **array** is defined in **mem.h** as follows:

```
struct dmachain {
  long address;  /* Physical address */
  short count;   /* Number of bytes */
};
```

Because contiguous virtual pages need not by physically contiguous, a number of these records, one for each underlying physical page, are written into **array**.

To address physical memory, use smem_create.

DIAGNOSTICS

vmtopm returns the number of physical mappings for the memory range if successful, or -1 if not.

If vmtopm fails, **errno** will contain one of the following:

EINVAL	The process ID **pid** is bad.
EFAULT	**array** is at a bad address.

NOTE

`vmtopm` may crash the system if called with the ID of a nonexistent process. Memory addressed physically is not guaranteed to remain in the same physical location, or even to be present anywhere, if the process is subject to virtual memory demand paging. To render a process immune to demand paging, run it at a higher priority than the priority at which demand paging was engaged, or do not engage virtual memory at all.

SEE ALSO

System Calls - `smem_create()`

NAME

`wait, waitpid, wait3` – wait for a child process to terminate or change status

SYNOPSIS

```
#include <wait.h>

pid_t wait(_ptr)
int *_ptr;

pid_t waitpid(pid, _ptr, options)
pid_t pid;
int *_ptr;
int options;

#include <time.h>
#include <resource.h>

pid_t wait3(_ptr, options, rusage)
int *_ptr;
int options;
struct rusage *rusage;

WIFSTOPPED(status)
int status;

WSTOPSIG(status)
int status;

WIFSIGNALED(status)
int status;

WTERMSIG(status)
int status;

WIFEXITED(status)
int status;

WEXITSTATUS(status)
int status;
```

DESCRIPTION

`wait` waits for a child process to terminate or for a traced child to stop. (A child process is "traced" if it has called the `ptrace` system call with a `request` argument of `0`.) When a child does terminate, or when a traced child stops, `wait` returns its process ID number, and the appropriate information about the child is written into the `wait` union pointed to by **`status`**. If the process has no children the call immediately returns `-1`.

`waitpid` behaves exactly like `wait` (except that it does not wait for traced children to stop) if **`pid`** is `-1` and **`options`** is `0`. Otherwise, it behaves in a modified manner depending on the values of **`pid`** and **`options`**. **`pid`** specifies a set of child processes on which to wait:

`pid` `= -1`	Wait for any child process.
`pid` `> 0`	Wait for the child process whose process ID is equal to **`pid`**.
`pid` `= 0`	Wait for any child process whose process group ID is equal to that of the calling process.
`pid` `< -1`	Wait for any child process whose process group ID is equal to the absolute value of **`pid`**.

Options may be the bitwise-OR of zero or more of the following flags:

`WNOHANG`	Do not suspend execution of the calling process if no children in the set specified by **`pid`** have terminated or stopped yet.
`WUNTRACED`	In addition to terminated child processes, include any stopped child process in the set specified by **`pid`** whose status has not already been reported since it stopped.

`waitpid` returns 0 if the `WNOHANG` option was specified and there are still children alive. It returns the **`pid`** of the last-exited child if there are no children left alive.

`wait3` is yet another way to wait for a child process. The **`options`** argument has the same meaning as in the `waitpid` system call (see above). `wait3` also allows resource use information for child processes to be collected. If **rusage** is non-NULL the resource use for the child and all its children is written to the structure pointed to by **rusage**. wait3 is not POSIX compliant.

The macros WIFSTOPPED, WIFSIGNALED, and WIFEXITED will evaluate to a non-zero value if the child process was stopped, killed by a signal, or exited willfully, respectively. If the child process was stopped or killed, the WSTOPSIG or WTERMSIG macro will evaluate to the signal responsible, respectively. If the child process exited by calling exit or _exit, the macro WEXITSTATUS will evaluate to the low 8 bits of the value passed to exit or _exit. Note that if the child process does not explicitly call exit or _exit and simply returns from main(), then exit is called automatically with the return value from main() as its argument.

wait3 returns 0 if the WNOHANG option was specified and there are no children to wait for.

DIAGNOSTICS

wait, waitpid, and wait3 return -1 if they fail.

Upon failure, **errno** will contain one of the following:

ECHILD	The calling process has no children.
EFAULT	Either **_ptr** or **rusage** does not point to allocated memory in the process' address space.

waitpid may also fail with the following **errno** value:

EINVAL	**options** has a bit set that is not WNOHANG or WUNTRACED.

SEE ALSO

System Calls - _exit()

NAME

`write`, `writev` – write to a file

SYNOPSIS

```
write(fd, buff, count)
int fd;
char *buff;
int count;
```

```
#include <uio.h>
writev(fd, iov, iovcnt)
int fd;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

`write` attempts to write **count** bytes from the buffer pointed to by **buff** to the file associated with the descriptor **fd**. `writev` allows several different buffers to be written in single system call. It gathers the output data from **iovcnt** buffers specified by the members of **iov** array.

Each **iovec** entry specifies the base address and length of a buffer:

```
struct iovec {
    caddr_t iovbase;          /* base address  */
    int iov_len;              /* size in bytes */
};
```

If the file is a regular file or a device capable of seeking, the seek pointer for the file is set to the file offset associated with **fd** and incremented as each character is written. The file offset is set to the end of the file if `O_APPEND` is set.

The actual number of characters written is returned. If there is not enough room on the physical device to accept all of the data then only the data that can fit will be written.

A write to a regular file will not return until the file status and file data have been physically written onto storage, if the `O_FSYNC` flag of the file status is set. Similarly, for the regular files, if the `O_DSYNC` flag is set, the write will not return until the file data has been physically written onto storage.

For STREAMS files the following exceptions apply:

- Associated with each `stream` is a minimum and maximum packet-size range. An attempt to write **`count`** bytes of **`buff`** to a `stream` where **`count`** falls outside the packet-size range will have one of two results.
  - If the minimum packet-size is zero then `write` will split the contents of **`buff`** into messages of length less than or equal to the maximum packet-size.
  - If the minimum packet-size is not zero then `write` will fail and `errno` will be set to `ERANGE`.
- The `O_NDELAY` flag effects different file types in different ways. The effect of the flag being set or cleared is outlined below:
  - When writing to a full pipe `write` will return zero if `O_NDELAY` is set; otherwise, `write` will wait until there is room in the pipe.
  - When writing to a stream that cannot accept data because of internal flow control conditions the `write` will fail if `O_NDELAY` is set; otherwise, `write` will wait until conditions allow it to complete. If the `stream` cannot accept more data after part of **`buff`** has been written and `O_NDELAY` is set then the number of bytes written so far will be returned.

DIAGNOSTICS

If `write` or `writev` fails, it returns -1.

If the call fails **`errno`** will contain one of the following:

`EAGAIN`	`O_NDELAY` is set and the `stream` cannot accept data.
`EBADF`	**`fd`** is not a valid file descriptor.
`EFAULT`	**`buff`** does not point to allocated memory in the process's address space.

EFBIG	The write would cause the size of the file to exceed the maximum file size.
EFTYPE	**fd** refers to a POSIX ipc special file.
EINTR	A signal interrupted the read call.
EINVAL	Tried to write to a stream linked to a multiplexer.
ENOLINK	
ENOSPC	No space left on the device where the file resides.
ENXIO	Hangup on stream associated with **fd**.
EPIPE	The write is to a pipe without an open read end.
ERANGE	**count** is outside the packet-size range for the stream and the minimum packet-size is not zero.

SEE ALSO

System Calls - dup(), execve(), fcntl(), flock(), lseek(), open(), pipe()

Calls for STREAMS special files include:

System Calls - getmsg(), putmsg(), poll()

NAME

`yield` – force a process to relinquish the CPU

SYNOPSIS

`void yield()`

DESCRIPTION

`yield` forces the currently running process to relinquish the CPU. It shall resume execution after all other runnable processes of equal or greater priority have been scheduled to run.

DIAGNOSTICS

`yield` has no return value.

CONFORMANCE

`yield` is implemented according to Draft 9 of the POSIX 1003.4 (Real-Time) standard.

SEE ALSO

Example - `yield.c`

Master Index

W

X-Z