man pages section 3: Realtime Library Functions

Part No: 816–5171–16
September 2010

ORACLE

100617@24378

Contents

Preface

Both novice users and those familar with the SunOS operating system can use online man pages to obtain information about the system and its features. A man page is intended to answer concisely the question "What does it do?" The man pages in general comprise a reference manual. They are not intended to be a tutorial.

Overview

The following contains a brief description of each man page section and the information it references:

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 6 contains available games and demos.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
- Section 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).
- Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer can include in a device driver.
- Section 9F describes the kernel functions available for use by device drivers.
- Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the `intro` pages for more information and detail about each section, and `man(1)` for more information about man pages in general.

NAME

This section gives the names of the commands or functions documented, followed by a brief description of what they do.

SYNOPSIS

This section shows the syntax of commands or functions. When a command or file does not exist in the standard path, its full path name is shown. Options and arguments are alphabetized, with single letter arguments first, and options with arguments next, unless a different argument order is required.

The following special characters are used in this section:

[ ] Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

. . . Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, "`filename ...`".

| Separator. Only one of the arguments separated by this character can be specified at a time.

{ } Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.

PROTOCOL

This section occurs only in subsection 3R to indicate the protocol description file.

DESCRIPTION

This section defines the functionality and behavior of the service. Thus it describes concisely what the command does. It does not discuss OPTIONS or cite EXAMPLES. Interactive commands, subcommands, requests, macros, and functions are described under USAGE.

IOCTL

This section appears on pages in Section 7 only. Only the device class that supplies appropriate parameters to the `ioctl(2)` system call is called `ioctl` and generates its own

	heading. `ioctl` calls for a specific device are listed alphabetically (on the man page for that specific device). `ioctl` calls are used for a particular class of devices all of which have an `io` ending, such as `mtio`(7I).
OPTIONS	This secton lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.
OPERANDS	This section lists the command operands and describes how they affect the actions of the command.
OUTPUT	This section describes the output – standard output, standard error, or output files – generated by the command.
RETURN VALUES	If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or –1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.
ERRORS	On failure, most functions place an error code in the global variable `errno` indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than one condition can cause the same error, each condition is described in a separate paragraph under the error code.
USAGE	This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality: Commands Modifiers Variables Expressions Input Grammar

EXAMPLES	This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%`, or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.
ENVIRONMENT VARIABLES	This section lists any environment variables that the command or function affects, followed by a brief description of the effect.
EXIT STATUS	This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion, and values other than zero for various error conditions.
FILES	This section lists all file names referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.
ATTRIBUTES	This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See `attributes(5)` for more information.
SEE ALSO	This section lists references to other man pages, in-house documentation, and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.
BUGS	This section describes known bugs and, wherever possible, suggests workarounds.

REFERENCE

Realtime Library Functions

Name aiocancel – cancel an asynchronous operation

Synopsis

```
cc [ flag ... ] file ... -laio [ library ... ]
#include <sys/asynch.h>


int aiocancel(aio_result_t *resultp);
```

Description `aiocancel()` cancels the asynchronous operation associated with the result buffer pointed to by *resultp*. It may not be possible to immediately cancel an operation which is in progress and in this case, `aiocancel()` will not wait to cancel it.

Upon successful completion, `aiocancel()` returns `0` and the requested operation is cancelled. The application will not receive the `SIGIO` completion signal for an asynchronous operation that is successfully cancelled.

Return Values Upon successful completion, `aiocancel()` returns `0`. Upon failure, `aiocancel()` returns `−1` and sets `errno` to indicate the error.

Errors `aiocancel()` will fail if any of the following are true:

`EACCES` The parameter *resultp* does not correspond to any outstanding asynchronous operation, although there is at least one currently outstanding.

`EFAULT` *resultp* points to an address outside the address space of the requesting process. See `NOTES`.

`EINVAL` There are not any outstanding requests to cancel.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `aioread(3AIO)`, `aiowait(3AIO)`, `attributes(5)`

Notes Passing an illegal address as *resultp* will result in setting `errno` to `EFAULT` *only* if it is detected by the application process.

Name aio_cancel – cancel asynchronous I/O request

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

Description The `aio_cancel()` function attempts to cancel one or more asynchronous I/O requests currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to the asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is `NULL`, then all outstanding cancelable asynchronous I/O requests against *fildes* are canceled.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. If there are requests that cannot be canceled, then the normal asynchronous completion process takes place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status is set to `ECANCELED` and the return status is −1. For requested operations that are not successfully canceled, the *aiocbp* is not modified by `aio_cancel()`.

If *aiocbp* is not `NULL`, then if *fildes* does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

Return Values The `aio_cancel()` function returns the value `AIO_CANCELED` to the calling process if the requested operation(s) were canceled. The value `AIO_NOTCANCELED` is returned if at least one of the requested operation(s) cannot be canceled because it is in progress. In this case, the state of the other operations, if any, referenced in the call to `aio_cancel()` is not indicated by the return value of `aio_cancel()`. The application may determine the state of affairs for these operations by using `aio_error`(3RT). The value `AIO_ALLDONE` is returned if all of the operations have already completed. Otherwise, the function returns −1 and sets `errno` to indicate the error.

Errors The `aio_cancel()` function will fail if:

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

`ENOSYS` The `aio_cancel()` function is not supported.

Usage The `aio_cancel()` function has a transitional interface for 64-bit file offsets. See `lf64`(5).

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also aio.h(3HEAD), signal.h(3HEAD), aio_read(3RT), aio_return(3RT), attributes(5), lf64(5), standards(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name aio_error – retrieve errors status for an asynchronous I/O operation

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);
```

Description The `aio_error()` function returns the error status associated with the `aiocb` structure referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the `errno` value that would be set by the corresponding `read(2)`, `write(2)`, or `fsync(3C)` operation. If the operation has not yet completed, then the error status will be equal to `EINPROGRESS`.

Return Values If the asynchronous I/O operation has completed successfully, then `0` is returned. If the asynchronous operation has completed unsuccessfully, then the error status, as described for `read(2)`, `write(2)`, and `fsync(3C)`, is returned. If the asynchronous I/O operation has not yet completed, then `EINPROGRESS` is returned.

Errors The `aio_error()` function will fail if:

`ENOSYS` The `aio_error()` function is not supported by the system.

The `aio_error()` function may fail if:

`EINVAL` The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

Usage The `aio_error()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Examples **EXAMPLE 1** The following is an example of an error handling routine using the `aio_error()` function.

```
#include <aio.h>
#include <errno.h>
#include <signal.h>
struct aiocb      my_aiocb;
struct sigaction  my_sigaction;
void              my_aio_handler(int, siginfo_t *, void *);
 . . .
my_sigaction.sa_flags = SA_SIGINFO;
my_sigaction.sa_sigaction = my_aio_handler;
sigemptyset(&my_sigaction.sa_mask);
(void) sigaction(SIGRTMIN, &my_sigaction, NULL);
 . . .
my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
my_aiocb.aio_sigevent.sigev_signo = SIGRTMIN;
my_aiocb.aio_sigevent.sigev_value.sival_ptr = &myaiocb;
 . . .
(void) aio_read(&my_aiocb);
 . . .
```

EXAMPLE 1 The following is an example of an error handling routine using the `aio_error()` function. *(Continued)*

```
void
my_aio_handler(int signo, siginfo_t *siginfo, void *context) {
int     my_errno;
struct aiocb    *my_aiocbp;

my_aiocbp = siginfo->si_value.sival_ptr;
        if ((my_errno = aio_error(my_aiocb)) != EINPROGRESS) {
                int my_status = aio_return(my_aiocb);
                if (my_status >= 0){ /* start another operation */
                         . . .
                } else        { /* handle I/O error */
                         . . .
                }
        }
}
```

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also

`_Exit(2)`, `close(2)`, `fork(2)`, `lseek(2)`, `read(2)`, `write(2)`, `aio.h(3HEAD)`, `aio_cancel(3RT)`, `aio_fsync(3RT)`, `aio_read(3RT)`, `aio_return(3RT)`, `aio_write(3RT)`, `lio_listio(3RT)`, `signal.h(3HEAD)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Notes

Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name aio_fsync – asynchronous file synchronization

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocbp);
```

Description The `aio_fsync()` function asynchronously forces all I/O operations associated with the file indicated by the file descriptor `aio_fildes` member of the `aiocb` structure referenced by the *aiocbp* argument and queued at the time of the call to `aio_fsync()` to the synchronized I/O completion state. The function call returns when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

If *op* is `O_DSYNC`, all currently queued I/O operations are completed as if by a call to `fdatasync`(3RT); that is, as defined for synchronized I/O data integrity completion. If *op* is `O_SYNC`, all currently queued I/O operations are completed as if by a call to `fsync`(3C); that is, as defined for synchronized I/O file integrity completion. If the `aio_fsync()` function fails, or if the operation queued by `aio_fsync()` fails, then, as for `fsync`(3C) and `fdatasync`(3RT), outstanding I/O operations are not guaranteed to have been completed.

If `aio_fsync()` succeeds, then it is only the I/O that was queued at the time of the call to `aio_fsync()` that is guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used as an argument to `aio_error`(3RT) and `aio_return`(3RT) in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation is `EINPROGRESS.` When all data has been successfully transferred, the error status will be reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation will be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to occur when all operations have achieved synchronized I/O completion. All other members of the structure referenced by *aiocbp* are ignored. If the control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If the `aio_fsync()` function fails or the *aiocbp* indicates an error condition, data is not guaranteed to have been successfully transferred.

If *aiocbp* is `NULL`, then no status is returned in *aiocbp*, and no signal is generated upon completion of the operation.

Return Values The aio_fsync() function returns 0 to the calling process if the I/O operation is successfully queued; otherwise, the function returns −1 and sets errno to indicate the error.

Errors The aio_fsync() function will fail if:

EAGAIN The requested asynchronous operation was not queued due to temporary resource limitations.

EBADF The aio_fildes member of the aiocb structure referenced by the *aiocbp* argument is not a valid file descriptor open for writing.

EINVAL The system does not support synchronized I/O for this file.

EINVAL A value of *op* other than O_DSYNC or O_SYNC was specified.

ENOSYS The aio_fsync() function is not supported by the system.

In the event that any of the queued I/O operations fail, aio_fsync() returns the error condition defined for read(2) and write(2). The error will be returned in the error status for the asynchronous fsync(3C) operation, which can be retrieved using aio_error(3RT).

Usage The aio_fsync() function has a transitional interface for 64-bit file offsets. See lf64(5).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also fcntl(2), open(2), read(2), write(2), aio_error(3RT), aio_return(3RT), fdatasync(3RT), fsync(3C), attributes(5), fcntl.h(3HEAD), aio.h(3HEAD), signal.h(3HEAD), attributes(5), lf64(5), standards(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name aioread, aiowrite – read or write asynchronous I/O operations

Synopsis

```
cc [ flag... ] file... -laio [ library... ]
#include <sys/types.h>
#include <sys/asynch.h>

int aioread(int fildes, char *bufp, int bufs, off_t offset,
     int whence, aio_result_t *resultp);

int aiowrite(int fildes, const char *bufp, int bufs, off_t offset,
     int whence, aio_result_t *resultp);
```

Description The `aioread()` function initiates one asynchronous `read(2)` and returns control to the calling program. The read continues concurrently with other activity of the process. An attempt is made to read *bufs* bytes of data from the object referenced by the descriptor *fildes* into the buffer pointed to by *bufp*.

The `aiowrite()` function initiates one asynchronous `write(2)` and returns control to the calling program. The write continues concurrently with other activity of the process. An attempt is made to write *bufs* bytes of data from the buffer pointed to by *bufp* to the object referenced by the descriptor *fildes*.

On objects capable of seeking, the I/O operation starts at the position specified by *whence* and *offset*. These parameters have the same meaning as the corresponding parameters to the `llseek(2)` function. On objects not capable of seeking the I/O operation always start from the current position and the parameters `whence` and *offset* are ignored. The seek pointer for objects capable of seeking is not updated by `aioread()` or `aiowrite()`. Sequential asynchronous operations on these devices must be managed by the application using the *whence* and *offset* parameters.

The result of the asynchronous operation is stored in the structure pointed to by *resultp*:

```
int aio_return;          /* return value of read() or write() */
int aio_errno;          /* value of errno for read() or write() */
```

Upon completion of the operation both `aio_return` and `aio_errno` are set to reflect the result of the operation. Since `AIO_INPROGRESS` is not a value used by the system, the client can detect a change in state by initializing `aio_return` to this value.

The application-supplied buffer *bufp* should not be referenced by the application until after the operation has completed. While the operation is in progress, this buffer is in use by the operating system.

Notification of the completion of an asynchronous I/O operation can be obtained synchronously through the `aiowait(3AIO)` function, or asynchronously by installing a signal handler for the `SIGIO` signal. Asynchronous notification is accomplished by sending the process a `SIGIO` signal. If a signal handler is not installed for the `SIGIO` signal, asynchronous notification is disabled. The delivery of this instance of the `SIGIO` signal is reliable in that a signal delivered while the handler is executing is not lost. If the client ensures that `aiowait()`

returns nothing (using a polling timeout) before returning from the signal handler, no asynchronous I/O notifications are lost. The `aiowait()` function is the only way to dequeue an asynchronous notification. The `SIGIO` signal can have several meanings simultaneously. For example, it can signify that a descriptor generated `SIGIO` and an asynchronous operation completed. Further, issuing an asynchronous request successfully guarantees that space exists to queue the completion notification.

The `close(2)`, `exit(2)` and `execve(2)`) functions block until all pending asynchronous I/O operations can be canceled by the system.

It is an error to use the same result buffer in more than one outstanding request. These structures can be reused only after the system has completed the operation.

Return Values Upon successful completion, `aioread()` and `aiowrite()` return `0`. Upon failure, `aioread()` and `aiowrite()` return −`1` and set `errno` to indicate the error.

Errors The `aioread()` and `aiowrite()` functions will fail if:

`EAGAIN` The number of asynchronous requests that the system can handle at any one time has been exceeded

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

`EFAULT` At least one of *bufp* or *resultp* points to an address outside the address space of the requesting process. This condition is reported only if detected by the application process.

`EINVAL` The *resultp* argument is currently being used by an outstanding asynchronous request.

`EINVAL` The *offset* argument is not a valid offset for this file system type.

`ENOMEM` Memory resources are unavailable to initiate request.

Usage The `aioread()` and `aiowrite()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `close(2)`, `execve(2)`, `exit(2)`, `llseek(2)`, `lseek(2)`, `open(2)`, `read(2)`, `write(2)`, `aiocancel(3AIO)`, `aiowait(3AIO)`, `sigvec(3UCB)`, `attributes(5)`, `lf64(5)`

Name aio_read – asynchronous read from a file

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

Description The `aio_read()` function allows the calling process to read *aiocbp*->`aio_nbytes` from the file associated with *aiocbp*->`aio_fildes` into the buffer pointed to by *aiocbp*->`aio_buf`. The function call returns when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately). If `_POSIX_PRIORITIZED_IO` is defined and prioritized I/O is supported for this file, then the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus *aiocbp*->`aio_reqprio`. The *aiocbp* value may be used as an argument to `aio_error(3RT)` and `aio_return(3RT)` in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. If an error condition is encountered during queuing, the function call returns without having initiated or queued the request. The requested operation takes place at the absolute position in the file as given by *aio_offset*, as if `lseek(2)` were called immediately prior to the operation with an *offset* equal to *aio_offset* and a `whence` equal to `SEEK_SET`. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aiocbp*->`aio_lio_opcode` field is ignored by `aio_read()`.

The *aiocbp* argument points to an `aiocb` structure. If the buffer pointed to by *aiocbp*->`aio_buf` or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

If `_POSIX_SYNCHRONIZED_IO` is defined and synchronized I/O is enabled on the file associated with *aiocbp*->`aio_fildes,` the behavior of this function is according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp*->`aio_fildes`.

Return Values The `aio_read()` function returns `0` to the calling process if the I/O operation is successfully queued; otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `aio_read()` function will fail if:

`EAGAIN` The requested asynchronous I/O operation was not queued due to system resource limitations.

`ENOSYS` The `aio_read()` function is not supported by the system.

Each of the following conditions may be detected synchronously at the time of the call to `aio_read()`, or asynchronously. If any of the conditions below are detected synchronously, the `aio_read()` function returns −1 and sets `errno` to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to −1, and the error status of the asynchronous operation will be set to the corresponding value.

`EBADF` — The *aiocbp*->`aio_fildes` argument is not a valid file descriptor open for reading.

`EINVAL` — The file offset value implied by *aiocbp*->`aio_offset` would be invalid, *aiocbp*->`aio_reqprio` is not a valid value, or *aiocbp*->`aio_nbytes` is an invalid value.

In the case that the `aio_read()` successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by the `read(2)` function call. In addition, the error status of the asynchronous operation will be set to one of the error statuses normally set by the `read()` function call, or one of the following values:

`EBADF` — The *aiocbp*->`aio_fildes` argument is not a valid file descriptor open for reading.

`ECANCELED` — The requested I/O was canceled before the I/O completed due to an explicit `aio_cancel(3RT)` request.

`EINVAL` — The file offset value implied by *aiocbp*->`aio_offset` would be invalid.

The following condition may be detected synchronously or asynchronously:

`EOVERFLOW` — The file is a regular file, *aiobcp*->`aio_nbytes` is greater than 0 and the starting offset in *aiobcp*->`aio_offset` is before the end-of-file and is at or beyond the offset maximum in the open file description associated with *aiocbp*->`aio_fildes`.

Usage

For portability, the application should set *aiocb*->`aio_reqprio` to `0`.

The `aio_read()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also close(2), exec(2), exit(2), fork(2), lseek(2), read(2), write(2), aio.h(3HEAD), siginfo.h(3HEAD), signal.h(3HEAD), aio_cancel(3RT), aio_return(3RT), lio_listio(3RT), attributes(5), lf64(5), standards(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name aio_return – retrieve return status of an asynchronous I/O operation

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

ssize_t aio_return(struct aiocb *aiocbp);
```

Description The `aio_return()` function returns the return status associated with the `aiocb` structure referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding `read(2)`, `write(2)`, or `fsync(3C)` function call. If the error status for the operation is equal to `EINPROGRESS`, then the return status for the operation is undefined. The `aio_return()` function may be called exactly once to retrieve the return status of a given asynchronous operation; thereafter, if the same `aiocb` structure is used in a call to `aio_return()` or `aio_error(3RT)`, an error may be returned. When the `aiocb` structure referred to by *aiocbp* is used to submit another asynchronous operation, then `aio_return()` may be successfully used to retrieve the return status of that operation.

Return Values If the asynchronous I/O operation has completed, then the return status, as described for `read(2)`, `write(2)`, and `fsync(3C)`, is returned. If the asynchronous I/O operation has not yet completed, the results of `aio_return()` are undefined.

Errors The `aio_return()` function will fail if:

`EINVAL` The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

`ENOSYS` The `aio_return()` function is not supported by the system.

Usage The `aio_return()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `close(2)`, `exec(2)`, `exit(2)`, `fork(2)`, `lseek(2)`, `read(2)`, `write(2)`, `fsync(3C)`, `aio.h(3HEAD)`, `signal.h(3HEAD)`, `aio_cancel(3RT)`, `aio_fsync(3RT)`, `aio_read(3RT)`, `lio_listio(3RT)`, `attributes(5)`, `lf64(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name aio_suspend – wait for asynchronous I/O request

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent,
     const struct timespec *timeout);
```

Description The `aio_suspend()` function suspends the calling thread until at least one of the asynchronous I/O operations referenced by the *list* argument has completed, until a signal interrupts the function, or, if *timeout* is not `NULL`, until the time interval specified by *timeout* has passed. If any of the `aiocb` structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to `EINPROGRESS`) at the time of the call, the function returns without suspending the calling thread. The *list* argument is an array of pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of elements in the array and is limited to `_AIO_LISTIO_MAX` = 4096. Each `aiocb` structure pointed to will have been used in initiating an asynchronous I/O request via `aio_read`(3RT), `aio_write`(3RT), or `lio_listio`(3RT). This array may contain null pointers, which are ignored. If this array contains pointers that refer to `aiocb` structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the `timespec` structure pointed to by *timeout* passes before any of the I/O operations referenced by *list* are completed, then `aio_suspend()` returns with an error.

Return Values If `aio_suspend()` returns after one or more asynchronous I/O operations have completed, it returns `0`. Otherwise, it returns −`1`, and sets `errno` to indicate the error.

The application may determine which asynchronous I/O completed by scanning the associated error and return status using `aio_error`(3RT) and `aio_return`(3RT), respectively.

Errors The `aio_suspend()` function will fail if:

`EAGAIN` No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by *timeout*.

`EINTR` A signal interrupted the `aio_suspend()` function. Since each asynchronous I/O operation might provoke a signal when it completes, this error return can be caused by the completion of one or more of the very I/O operations being awaited.

`EINVAL` The *nent* argument is less than or equal to 0 or greater than `_AIO_LISTIO_MAX`, or the `timespec` structure pointed to by *timeout* is not properly set because *tv_sec* is less than 0 or *tv_nsec* is either less than 0 or greater than 10^9.

`ENOMEM` There is currently not enough available memory; the application can try again later.

`ENOSYS` The `aio_suspend()` function is not supported by the system.

Usage The aio_suspend() function has a transitional interface for 64-bit file offsets. See lf64(5).

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also aio.h(3HEAD), aio_fsync(3RT), aio_read(3RT), aio_return(3RT), aio_write(3RT), lio_listio(3RT), signal.h(3HEAD), attributes(5), lf64(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name aiowait – wait for completion of asynchronous I/O operation

Synopsis

```
cc [ flag... ] file... -laio [ library... ]
#include <sys/asynch.h>
#include <sys/time.h>

aio_result_t *aiowait(const struct timeval *timeout);
```

Description The `aiowait()` function suspends the calling process until one of its outstanding asynchronous I/O operations completes, providing a synchronous method of notification.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the completion of an asynchronous I/O operation. If *timeout* is a zero pointer, `aiowait()` blocks indefinitely. To effect a poll, the *timeout* parameter should be non-zero, pointing to a zero-valued *timeval* structure.

The `timeval` structure is defined in `<sys/time.h>` and contains the following members:

```
long  tv_sec;          /* seconds */
long  tv_usec;         /* and microseconds */
```

Return Values Upon successful completion, `aiowait()` returns a pointer to the result structure used when the completed asynchronous I/O operation was requested. Upon failure, `aiowait()` returns −1 and sets `errno` to indicate the error. `aiowait()` returns `0` if the time limit expires.

Errors The `aiowait()` function will fail if:

`EFAULT` The *timeout* argument points to an address outside the address space of the requesting process. See `NOTES`.

`EINTR` The execution of `aiowait()` was interrupted by a signal.

`EINVAL` There are no outstanding asynchronous I/O requests.

`EINVAL` The `tv_secs` member of the `timeval` structure pointed to by *timeout* is less than 0 or the `tv_usecs` member is greater than the number of seconds in a microsecond.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT-Level	Safe

See Also `aiocancel(3AIO)`, `aioread(3AIO)`, `attributes(5)`

Notes The `aiowait()` function is the only way to dequeue an asynchronous notification. It can be used either inside a `SIGIO` signal handler or in the main program. One `SIGIO` signal can represent several queued events.

Passing an illegal address as *timeout* will result in setting `errno` to `EFAULT` only if detected by the application process.

Name aio_waitn – wait for completion of asynchronous I/O operations

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

int aio_waitn(struct aiocb *list[], uint_t nent,
     uint_t *nwait, const struct timespec *timeout);
```

Description The `aio_waitn()` function suspends the calling thread until at least the number of requests specified by *nwait* have completed, until a signal interrupts the function, or if *timeout* is not `NULL`, until the time interval specified by *timeout* has passed.

To effect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued `timespec` structure.

The *list* argument is an array of uninitialized I/O completion block pointers to be filled in by the system before `aio_waitn()` returns. The *nent* argument indicates the maximum number of elements that can be placed in *list*[] and is limited to `_AIO_LISTIO_MAX` = 4096.

The *nwait* argument points to the minimum number of requests `aio_waitn()` should wait for. Upon returning, the content of *nwait* is set to the actual number of requests in the `aiocb` list, which can be greater than the initial value specified in *nwait*. The `aio_waitn()` function attempts to return as many requests as possible, up to the number of outstanding asynchronous I/Os but less than or equal to the maximum specified by the *nent* argument. As soon as the number of outstanding asynchronous I/O requests becomes 0, `aio_waitn()` returns with the current list of completed requests.

The `aiocb` structures returned will have been used in initiating an asynchronous I/O request from any thread in the process with `aio_read(3RT)`, `aio_write(3RT)`, or `lio_listio(3RT)`.

If the time interval expires before the expected number of I/O operations specified by *nwait* are completed, `aio_waitn()` returns the number of completed requests and the content of the *nwait* pointer is updated with that number.

If `aio_waitn()` is interrupted by a signal, *nwait* is set to the number of completed requests.

The application can determine the status of the completed asynchronous I/O by checking the associated error and return status using `aio_error(3RT)` and `aio_return(3RT)`, respectively.

Return Values Upon successful completion, `aio_waitn()` returns 0. Otherwise, it returns `-1` and sets `errno` to indicate the error.

Errors The `aio_waitn()` function will fail if:

`EAGAIN` There are no outstanding asynchronous I/O requests.

`EFAULT` The *list*[], *nwait*, or *timeout* argument points to an address outside the address space of the process. The `errno` variable is set to `EFAULT` only if this condition is detected by the application process.

EINTR The execution of `aio_waitn()` was interrupted by a signal.

EINVAL The *timeout* element *tv_sec* or *tv_nsec* is < 0, *nent* is set to 0 or > `_AIO_LISTIO_MAX`, or *nwait* is either set to 0 or is > *nent*.

ENOMEM There is currently not enough available memory. The application can try again later.

ETIME The time interval expired before *nwait* outstanding requests have completed.

Usage The `aio_waitn()` function has a transitional interface for 64-bit file offsets. See `lf64`(5).

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Stable
MT-Level	Safe

See Also `aio.h`(3HEAD), `aio_error`(3RT), `aio_read`(3RT), `aio_write`(3RT), `lio_listio`(3RT), `aio_return`(3RT), `attributes`(5), `lf64`(5)

Name aio_write – asynchronous write to a file

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

int aio_write(struct aiocb *aiocbp);
```

Description The `aio_write()` function allows the calling process to write *aiocbp*->`aio_nbytes` to the file associated with *aiocbp*->`aio_fildes` from the buffer pointed to by *aiocbp*->`aio_buf`. The function call returns when the write request has been initiated or, at a minimum, queued to the file or device. If `_POSIX_PRIORITIZED_IO` is defined and prioritized I/O is supported for this file, then the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus *aiocbp*->`aio_reqprio`. The *aiocbp* may be used as an argument to `aio_error(3RT)` and `aio_return(3RT)` in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding.

The *aiocbp* argument points to an `aiocb` structure. If the buffer pointed to by *aiocbp*->`aio_buf` or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If `O_APPEND` is not set for the file descriptor *aio_fildes*, then the requested operation takes place at the absolute position in the file as given by *aio_offset*, as if `lseek(2)` were called immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to `SEEK_SET`. If `O_APPEND` is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aiocbp*->`aio_lio_opcode` field is ignored by `aio_write()`.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

If `_POSIX_SYNCHRONIZED_IO` is defined and synchronized I/O is enabled on the file associated with *aiocbp*->`aio_fildes`, the behavior of this function shall be according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp*->`aio_fildes`.

Return Values The `aio_write()` function returns `0` to the calling process if the I/O operation is successfully queued; otherwise, the function returns −1 and sets `errno` to indicate the error.

Errors The `aio_write()` function will fail if:

`EAGAIN` The requested asynchronous I/O operation was not queued due to system resource limitations.

ENOSYS The `aio_write()` function is not supported by the system.

Each of the following conditions may be detected synchronously at the time of the call to `aio_write()`, or asynchronously. If any of the conditions below are detected synchronously, the `aio_write()` function returns −1 and sets `errno` to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to −1, and the error status of the asynchronous operation will be set to the corresponding value.

EBADF The *aiocbp*->`aio_fildes` argument is not a valid file descriptor open for writing.

EINVAL The file offset value implied by *aiocbp*->`aio_offset` would be invalid, *aiocbp*->`aio_reqprio` is not a valid value, or *aiocbp*->`aio_nbytes` is an invalid value.

In the case that the `aio_write()` successfully queues the I/O operation, the return status of the asynchronous operation will be one of the values normally returned by the `write(2)` function call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the `write()` function call, or one of the following:

EBADF The *aiocbp*->`aio_fildes` argument is not a valid file descriptor open for writing.

EINVAL The file offset value implied by *aiocbp*->`aio_offset` would be invalid.

ECANCELED The requested I/O was canceled before the I/O completed due to an explicit `aio_cancel(3RT)` request.

The following condition may be detected synchronously or asynchronously:

EFBIG The file is a regular file, *aiobcp*->`aio_nbytes` is greater than 0 and the starting offset in *aiobcp*->`aio_offset` is at or beyond the offset maximum in the open file description associated with *aiocbp*->`aio_fildes`.

Usage The `aio_write()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also aio_cancel(3RT), aio_error(3RT), aio_read(3RT), aio_return(3RT), lio_listio(3RT), close(2), _Exit(2), fork(2), lseek(2), write(2), aio.h(3HEAD), signal.h(3HEAD), attributes(5), lf64(5), standards(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name clock_nanosleep – high resolution sleep with specifiable clock

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
     const struct timespec *rqtp, struct timespec *rmtp);
```

Description If the flag `TIMER_ABSTIME` is not set in the *flags* argument, the `clock_nanosleep()` function causes the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. The clock used to measure the time is the clock specified by *clock_id*.

If the flag `TIMER_ABSTIME` is set in the *flags* argument, the `clock_nanosleep()` function causes the current thread to be suspended from execution until either the time value of the clock specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. If, at the time of the call, the time value specified by *rqtp* is less than or equal to the time value of the specified clock, then `clock_nanosleep()` returns immediately and the calling process is not suspended.

The suspension time caused by this function can be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time for the relative `clock_nanosleep()` function (that is, with the `TIMER_ABSTIME` flag not set) will not be less than the time interval specified by rqtp, as measured by the corresponding clock. The suspension for the absolute `clock_nanosleep()` function (that is, with the `TIMER_ABSTIME` flag set) will be in effect at least until the value of the corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being interrupted by a signal.

The use of the `clock_nanosleep()` function has no effect on the action or blockage of any signal.

The `clock_nanosleep()` function fails if the *clock_id* argument refers to the CPU-time clock of the calling thread. It is unspecified if *clock_id* values of other CPU-time clocks are allowed.

Return Values If the `clock_nanosleep()` function returns because the requested time has elapsed, its return value is 0.

If the `clock_nanosleep()` function returns because it has been interrupted by a signal, it returns the corresponding error value. For the relative `clock_nanosleep()` function, if the *rmtp* argument is non-null, the timespec structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the remaining time is not returned. The absolute `clock_nanosleep()` function has no effect on the structure referenced by *rmtp*.

If `clock_nanosleep()` fails, it shall return the corresponding error value.

Errors The `clock_nanosleep()` function will fail if:

`EINTR` The `clock_nanosleep()` function was interrupted by a signal.

`EINVAL` The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1,000 million; or the TIMER_ABSTIME flag was specified in *flags* and the *rqtp* argument is outside the range for the clock specified by *clock_id*; or the *clock_id* argument does not specify a known clock, or specifies the CPU-time clock of the calling thread.

`ENOTSUP` The *clock_id* argument specifies a clock for which `clock_nanosleep()` is not supported, such as a CPU-time clock.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `clock_getres(3RT)`, `nanosleep(3RT)`, `pthread_cond_timedwait(3C)`, `sleep(3C)`, `attributes(5)`, `standards(5)`

Name clock_settime, clock_gettime, clock_getres – high-resolution clock operations

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <time.h>

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

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

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

Description The `clock_settime()` function sets the specified clock, *clock_id*, to the value specified by *tp*. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock are truncated down to the smaller multiple of the resolution.

The `clock_gettime()` function returns the current value *tp* for the specified clock, *clock_id*.

The resolution of any clock can be obtained by calling `clock_getres()`. Clock resolutions are system-dependent and cannot be set by a process. If the argument *res* is not `NULL`, the resolution of the specified clock is stored in the location pointed to by *res*. If *res* is `NULL`, the clock resolution is not returned. If the time argument of `clock_settime()` is not a multiple of *res*, then the value is truncated to a multiple of *res*.

A clock may be systemwide (that is, visible to all processes) or per-process (measuring time that is meaningful only within a process).

A *clock_id* of `CLOCK_REALTIME` is defined in `<time.h>`. This clock represents the realtime clock for the system. For this clock, the values returned by `clock_gettime()` and specified by `clock_settime()` represent the amount of time (in seconds and nanoseconds) since the Epoch. Additional clocks may also be supported. The interpretation of time values for these clocks is unspecified.

A *clock_id* of `CLOCK_HIGHRES` represents the non-adjustable, high-resolution clock for the system. For this clock, the value returned by `clock_gettime(3RT)` represents the amount of time (in seconds and nanoseconds) since some arbitrary time in the past; it is not correlated in any way to the time of day, and thus is not subject to resetting or drifting by way of `adjtime(2)`, `ntp_adjtime(2)`, `settimeofday(3C)`, or `clock_settime()`. The time source for this clock is the same as that for `gethrtime(3C)`.

Additional clocks may also be supported. The interpretation of time values for these clocks is unspecified.

Return Values Upon successful completion, `0` is returned. Otherwise, `−1` is returned and `errno` is set to indicate the error.

Errors The `clock_settime()`, `clock_gettime()` and `clock_getres()` functions will fail if:

`EINVAL` The *clock_id* argument does not specify a known clock.

`ENOSYS` The functions `clock_settime()`, `clock_gettime()`, and `clock_getres()` are not supported by this implementation.

The `clock_settime()` function will fail if:

`EINVAL` The *tp* argument to `clock_settime()` is outside the range for the given clock ID; or the *tp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

The `clock_settime()` function may fail if:

`EPERM` The requesting process does not have the appropriate privilege to set the specified clock.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	`clock_gettime()` is Async-Signal-Safe

See Also `time(2)`, `ctime(3C)`, `gethrtime(3C)`, `time.h(3HEAD)`, `timer_gettime(3RT)`, `attributes(5)`, `standards(5)`

Name door_bind, door_unbind – bind or unbind the current thread with the door server pool

Synopsis

```
cc -mt [ flag... ] file... -ldoor  [ library... ]
#include <door.h>

int door_bind(int did);

int door_unbind(void);
```

Description The `door_bind()` function associates the current thread with a door server pool. A door server pool is a private pool of server threads that is available to serve door invocations associated with the door *did*.

The `door_unbind()` function breaks the association of `door_bind()` by removing any private door pool binding that is associated with the current thread.

Normally, door server threads are placed in a global pool of available threads that invocations on any door can use to dispatch a door invocation. A door that has been created with `DOOR_PRIVATE` only uses server threads that have been associated with the door by `door_bind()`. It is therefore necessary to bind at least one server thread to doors created with `DOOR_PRIVATE`.

The server thread create function, `door_server_create()`, is initially called by the system during a `door_create()` operation. See `door_server_create(3DOOR)` and `door_create(3DOOR)`.

The current thread is added to the private pool of server threads associated with a door during the next `door_return()` (that has been issued by the current thread after an associated `door_bind()`). See `door_return(3DOOR)`. A server thread performing a `door_bind()` on a door that is already bound to a different door performs an implicit `door_unbind()` of the previous door.

If a process containing threads that have been bound to a door calls `fork(2)`, the threads in the child process will be bound to an invalid door, and any calls to `door_return(3DOOR)` will result in an error.

Return Values Upon successful completion, a `0` is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `door_bind()` and `door_unbind()` functions fail if:

`EBADF`	The *did* argument is not a valid door.
`EBADF`	The `door_unbind()` function was called by a thread that is currently not bound.
`EINVAL`	*did* was not created with the `DOOR_PRIVATE` attribute.

Examples **EXAMPLE 1** Use `door_bind()` to create private server pools for two doors.

The following example shows the use of `door_bind()` to create private server pools for two doors, `d1` and `d2`. Function `my_create()` is called when a new server thread is needed; it creates a thread running function, `my_server_create()`, which binds itself to one of the two doors.

```
#include <door.h>
#include <thread.h>
#include <pthread.h>
thread_key_t door_key;
int d1 = -1;
int d2 = -1;
cond_t cv;       /* statically initialized to zero */
mutex_t lock;    /* statically initialized to zero */

extern void foo(void *, char *, size_t, door_desc_t *, uint_t);
extern void bar(void *, char *, size_t, door_desc_t *, uint_t);

static void *
my_server_create(void *arg)
{
        /* wait for d1 & d2 to be initialized */
        mutex_lock(&lock);
        while (d1 == -1 || d2 == -1)
                cond_wait(&cv, &lock);
        mutex_unlock(&lock);

        if (arg == (void *)foo){
                /* bind thread with pool associated with d1 */
                thr_setspecific(door_key, (void *)foo);
                if (door_bind(d1) < 0) {
                        perror("door_bind"); exit (-1);
                }
        } else if (arg == (void *)bar) {
                /* bind thread with pool associated with d2 */
                thr_setspecific(door_key, (void *)bar);
                if (door_bind(d2) < 0) {
                /* bind thread to d2 thread pool */
                        perror("door_bind"); exit (-1);
                }
        }
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        door_return(NULL, 0, NULL, 0);  /* Wait for door invocation */
}

static void
my_create(door_info_t *dip)
```

EXAMPLE 1 Use `door_bind()` to create private server pools for two doors. *(Continued)*

```
{
        /* Pass the door identity information to create function */
        thr_create(NULL, 0, my_server_create, (void *)dip->di_proc,
                THR_BOUND | THR_DETACHED, NULL);
}

main()
{
        (void) door_server_create(my_create);
        if (thr_keycreate(&door_key, NULL) != 0) {
                perror("thr_keycreate");
                exit(1);
        }
        mutex_lock(&lock);
        d1 = door_create(foo, NULL, DOOR_PRIVATE); /* Private pool */
        d2 = door_create(bar, NULL, DOOR_PRIVATE); /* Private pool */
        cond_signal(&cv);
        mutex_unlock(&lock);
        while (1)
                pause();
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Evolving
MT-Level	Safe

See Also

fork(2),door_create(3DOOR), door_return(3DOOR), door_server_create(3DOOR), libdoor(3LIB), attributes(5)

Name door_call – invoke the function associated with a door descriptor

Synopsis `cc [ flag... ] file... -ldoor [ library... ]`

```
#include <door.h>

typedef struct {
     char            *data_ptr;     /* Argument/result buf ptr*/
     size_t          data_size;     /* Argument/result buf size */
     door_desc_t     *desc_ptr;     /* Argument/result descriptors */
     uint_t          desc_num;      /* Argument/result num desc */
     char            *rbuf;         /* Result buffer */
     size_t          rsize;         /* Result buffer size */
} door_arg_t;

int door_call(int d, door_arg_t *params);
```

Description The `door_call()` function invokes the function associated with the door descriptor *d*, and passes the arguments (if any) specified in *params*. All of the *params* members are treated as in/out parameters during a door invocation and may be updated upon returning from a door call. Passing `NULL` for *params* indicates there are no arguments to be passed and no results expected.

Arguments are specified using the `data_ptr` and `desc_ptr` members of *params*. The size of the argument data in bytes is passed in `data_size` and the number of argument descriptors is passed in `desc_num`.

Results from the door invocation are placed in the buffer, `rbuf`. See `door_return`(3DOOR). The `data_ptr` and `desc_ptr` members of *params* are updated to reflect the location of the results within the `rbuf` buffer. The size of the data results and number of descriptors returned are updated in the `data_size` and `desc_num` members. It is acceptable to use the same buffer for input argument data and results, so `door_call()` may be called with `data_ptr` and `desc_ptr` pointing to the buffer `rbuf`.

If the results of a door invocation exceed the size of the buffer specified by `rsize`, the system automatically allocates a new buffer in the caller's address space and updates the `rbuf` and `rsize` members to reflect this location. In this case, the caller is responsible for reclaiming this area using `munmap(rbuf, rsize)` when the buffer is no longer required. See `munmap`(2).

Descriptors passed in a `door_desc_t` structure are identified by the `d_attributes` member. The client marks the `d_attributes` member with the type of object being passed by logically OR-ing the value of object type. Currently, the only object type that can be passed or returned is a file descriptor, denoted by the `DOOR_DESCRIPTOR` attribute. Additionally, the `DOOR_RELEASE` attribute can be set, causing the descriptor to be closed in the caller's address space after it is passed to the target. The descriptor will be closed even if `door_call()` returns an error, unless that error is `EFAULT` or `EBADF`.

The `door_desc_t` structure includes the following members:

```
typedef struct {
        door_attr_t d_attributes;   /* Describes the parameter */
        union {
                struct {
                        int d_descriptor;   /* Descriptor */
                        door_id_t d_id;    /* Unique door id */
                        } d_desc;
                } d_data;
        } door_desc_t;
```

When file descriptors are passed or returned, a new descriptor is created in the target address space and the `d_descriptor` member in the target argument is updated to reflect the new descriptor. In addition, the system passes a system-wide unique number associated with each door in the `door_id` member and marks the `d_attributes` member with other attributes associated with a door including the following:

`DOOR_LOCAL` The door received was created by this process using `door_create()`. See `door_create`(3DOOR).

`DOOR_PRIVATE` The door received has a private pool of server threads associated with the door.

`DOOR_UNREF` The door received is expecting an unreferenced notification.

`DOOR_UNREF_MULTI` Similar to `DOOR_UNREF`, except multiple unreferenced notifications may be delivered for the same door.

`DOOR_REFUSE_DESC` This door does not accept argument descriptors.

`DOOR_REVOKED` The door received has been revoked by the server.

The `door_call()` function is not a restartable system call. It returns `EINTR` if a signal was caught and handled by this thread. If the door invocation is not idempotent the caller should mask any signals that may be generated during a `door_call()` operation. If the client aborts in the middle of a `door_call()`, the server thread is notified using the POSIX (see `standards`(5)) thread cancellation mechanism. See `cancellation`(5).

The descriptor returned from `door_create()` is marked as close on `exec (FD_CLOEXEC)`. Information about a door is available for all clients of a door using `door_info()`. Applications concerned with security should not place secure information in door data that is accessible by `door_info()`. In particular, secure data should not be stored in the data item *cookie*. See `door_info`(3DOOR).

Return Values Upon successful completion, `0` is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors The `door_call()` function will fail if:

`E2BIG` Arguments were too big for server thread stack.

EAGAIN	Server was out of available resources.
EBADF	Invalid door descriptor was passed.
EFAULT	Argument pointers pointed outside the allocated address space.
EINTR	A signal was caught in the client, the client called fork(2), or the server exited during invocation.
EINVAL	Bad arguments were passed.
EMFILE	The client or server has too many open descriptors.
ENOTSUP	The *desc_num* argument is non-zero and the door has the DOOR_REFUSE_DESC flag set.
EOVERFLOW	System could not create overflow area in caller for results.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Evolving
MT-Level	Safe

See Also munmap(2), door_create(3DOOR), door_info(3DOOR), door_return(3DOOR), libdoor(3LIB), attributes(5), cancellation(5), standards(5)

Name door_create – create a door descriptor

Synopsis

```
cc -mt [ flag ... ] file ... -ldoor  [ library ... ]
#include <door.h>

int door_create(void (*server_procedure) (void *cookie,
     char *argp, size_t arg_size, door_desc_t *dp, uint_t n_desc),
     void *cookie, uint_t attributes);
```

Description The `door_create()` function creates a door descriptor that describes the procedure specified by the function *server_procedure*. The data item, *cookie*, is associated with the door descriptor, and is passed as an argument to the invoked function *server_procedure* during `door_call`(3DOOR) invocations. Other arguments passed to *server_procedure* from an associated `door_call()` are placed on the stack and include *argp* and *dp*. The *argp* argument points to *arg_size* bytes of data and the *dp* argument points to *n_desc* `door_desc_t` structures. The *attributes* argument specifies attributes associated with the newly created door. Valid values for *attributes* are constructed by OR-ing one or more of the following values:

`DOOR_UNREF`
: Delivers a special invocation on the door when the number of descriptors that refer to this door drops to one. In order to trigger this condition, more than one descriptor must have referred to this door at some time. `DOOR_UNREF_DATA` designates an unreferenced invocation, as the *argp* argument passed to *server_procedure*. In the case of an unreferenced invocation, the values for *arg_size*, *dp* and *n_did* are `0`. Only one unreferenced invocation is delivered on behalf of a door.

`DOOR_UNREF_MULTI`
: Similar to `DOOR_UNREF`, except multiple unreferenced invocations can be delivered on the same door if the number of descriptors referring to the door drops to one more than once. Since an additional reference may have been passed by the time an unreferenced invocation arrives, the `DOOR_IS_UNREF` attribute returned by the `door_info`(3DOOR) call can be used to determine if the door is still unreferenced.

`DOOR_PRIVATE`
: Maintains a separate pool of server threads on behalf of the door. Server threads are associated with a door's private server pool using `door_bind`(3DOOR). See also `door_xcreate`(3DOOR) for an alternative means of creating private doors.

`DOOR_REFUSE_DESC`
: Any attempt to `door_call`(3DOOR) this door with argument descriptors will fail with `ENOTSUP`. When this flag is set, the door's server procedure will always be invoked with an *n_desc* argument of 0.

The descriptor returned from `door_create()` will be marked as close on exec (`FD_CLOEXEC`). Information about a door is available for all clients of a door using `door_info`(3DOOR). Applications concerned with security should not place secure information in door data that is accessible by `door_info()`. In particular, secure data should not be stored in the data item *cookie*.

By default, additional threads are created as needed to handle concurrent door_call(3DOOR) invocations. See door_server_create(3DOOR) for information on how to change this behavior.

A process can advertise a door in the file system name space using fattach(3C).

Return Values Upon successful completion, door_create() returns a non-negative value. Otherwise, door_create returns −1 and sets errno to indicate the error.

Errors The door_create() function will fail if:

EINVAL	Invalid attributes are passed.
EMFILE	The process has too many open descriptors.

Examples **EXAMPLE 1** Create a door and use fattach() to advertise the door in the file system namespace.

The following example creates a door and uses fattach() to advertise the door in the file system namespace.

```
void
server(void *cookie, char *argp, size_t arg_size, door_desc_t *dp,
    uint_t n_desc)
{
    door_return(NULL, 0, NULL, 0);
    /* NOTREACHED */
}

int
main(int argc, char *argv[])
{
    int did;
    struct stat buf;

    if ((did = door_create(server, 0, 0)) < 0) {
        perror("door_create");
        exit(1);
    }

    /* make sure file system location exists */
    if (stat("/tmp/door", &buf) < 0) {
        int newfd;
        if ((newfd = creat("/tmp/door", 0444)) < 0) {
            perror("creat");
            exit(1);
        }
        (void) close(newfd);
    }
```

EXAMPLE 1 Create a door and use `fattach()` to advertise the door in the file system namespace. *(Continued)*

```
    /* make sure nothing else is attached */
    (void) fdetach("/tmp/door");

    /* attach to file system */
    if (fattach(did, "/tmp/door") < 0) {
        perror("fattach");
        exit(2);
    }
    [...]
}
```

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Evolving
MT-Level	Safe

See Also `door_bind(3DOOR)`, `door_call(3DOOR)`, `door_info(3DOOR)`, `door_revoke(3DOOR)`, `door_server_create(3DOOR)`, `door_xcreate(3DOOR)`, `fattach(3C)`, `libdoor(3LIB)`, `attributes(5)`

Name door_cred – return credential information associated with the client

Synopsis

```
cc -mt [ flag ... ] file ... -ldoor  [ library ... ]
#include <door.h>

int door_cred(door_cred_t *info);
```

Description The `door_cred()` function returns credential information associated with the client (if any) of the current door invocation.

The contents of the *info* argument include the following fields:

```
uid_t   dc_euid;        /* Effective uid of client */
gid_t   dc_egid;        /* Effective gid of client */
uid_t   dc_ruid;        /* Real uid of client */
gid_t   dc_rgid;        /* Real gid of client */
pid_t   dc_pid;         /* pid of client */
```

The credential information associated with the client refers to the information from the immediate caller; not necessarily from the first thread in a chain of door calls.

Return Values Upon successful completion, `door_cred()` returns `0`. Otherwise, `door_cred()` returns −1 and sets `errno` to indicate the error.

Errors The `door_cred()` function will fail if:

`EFAULT` The address of the *info* argument is invalid.

`EINVAL` There is no associated door client.

Usage The `door_cred()` function is obsolete. Applications should use the `door_ucred`(3DOOR) function in place of `door_cred()`.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Obsolete
MT-Level	Safe

See Also `door_call`(3DOOR), `door_create`(3DOOR), `door_ucred`(3DOOR), `libdoor`(3LIB), `attributes`(5)

Name door_info – return information associated with a door descriptor

Synopsis

```
cc [ flag ... ] file ... -ldoor [ library ... ]
#include <door.h>

int door_info(int d, struct door_info *info);
```

Description The `door_info()` function returns information associated with a door descriptor. It obtains information about the door descriptor *d* and places the information that is relevant to the door in the structure pointed to by the *info* argument.

The `door_info` structure pointed to by the *info* argument contains the following members:

```
pid_t          di_target;      /* door server pid */
door_ptr_t     di_proc;        /* server function */
door_ptr_t     di_data;        /* data cookie for invocation */
door_attr_t    di_attributes;  /* door attributes */
door_id_t      di_uniquifier;  /* unique id among all doors */
```

The `di_target` member is the process ID of the door server, or −1 if the door server process has exited.

The values for `di_attributes` may be composed of the following:

`DOOR_LOCAL`	The door descriptor refers to a service procedure in this process.
`DOOR_UNREF`	The door has requested notification when all but the last reference has gone away.
`DOOR_UNREF_MULTI`	Similar to `DOOR_UNREF`, except multiple unreferenced notifications may be delivered for this door.
`DOOR_IS_UNREF`	There is currently only one descriptor referring to the door.
`DOOR_REFUSE_DESC`	The door refuses any attempt to `door_call`(3DOOR) it with argument descriptors.
`DOOR_REVOKED`	The door descriptor refers to a door that has been revoked.
`DOOR_PRIVATE`	The door has a separate pool of server threads associated with it.

The `di_proc` and `di_data` members are returned as `door_ptr_t` objects rather than `void *` pointers to allow clients and servers to interoperate in environments where the pointer sizes may vary in size (for example, 32-bit clients and 64-bit servers). Each door has a system-wide unique number associated with it that is set when the door is created by `door_create()`. This number is returned in `di_uniquifier`.

Return Values Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors The door_info() function will fail if:

EFAULT The address of argument *info* is an invalid address.

EBADF *d* is not a door descriptor.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Evolving
MT-Level	Safe

See Also door_bind(3DOOR), door_call(3DOOR), door_create(3DOOR), door_server_create(3DOOR), libdoor(3LIB), attributes(5)

Name door_return – return from a door invocation

Synopsis

```
cc -mt [ flag ... ] file ... -ldoor  [ library ... ]
#include <door.h>

int door_return(char *data_ptr, size_t data_size, door_desc_t *desc_ptr,
     uint_t num_desc);
```

Description The `door_return()` function returns from a door invocation. It returns control to the thread that issued the associated `door_call()` and blocks waiting for the next door invocation. See `door_call(3DOOR)`. Results, if any, from the door invocation are passed back to the client in the buffers pointed to by *data_ptr* and *desc_ptr*. If there is not a client associated with the `door_return()`, the calling thread discards the results, releases any passed descriptors with the `DOOR_RELEASE` attribute, and blocks waiting for the next door invocation.

Return Values Upon successful completion, `door_return()` does not return to the calling process. Otherwise, `door_return()` returns −1 to the calling process and sets `errno` to indicate the error.

Errors The `door_return()` function fails and returns to the calling process if:

`E2BIG` Arguments were too big for client.

`EFAULT` The address of *data_ptr* or *desc_ptr* is invalid.

`EINVAL` Invalid `door_return()` arguments were passed or a thread is bound to a door that no longer exists.

`EMFILE` The client has too many open descriptors.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Evolving
MT-Level	Safe

See Also `door_call(3DOOR)`,`libdoor(3LIB)`, `attributes(5)`

Name door_revoke – revoke access to a door descriptor

Synopsis

```
cc -mt [ flag ... ] file ... -ldoor  [ library ... ]
#include <door.h>

int door_revoke(int d);
```

Description The `door_revoke()` function revokes access to a door descriptor. Door descriptors are created with `door_create`(3DOOR). The `door_revoke()` function performs an implicit call to `close`(2), marking the door descriptor *d* as invalid.

A door descriptor can only be revoked by the process that created it. Door invocations that are in progress during a `door_revoke()` invocation are allowed to complete normally.

Return Values Upon successful completion, `door_revoke()` returns `0`. Otherwise, `door_revoke()` returns `−1` and sets `errno` to indicate the error.

Errors The `door_revoke()` function will fail if:

`EBADF` An invalid door descriptor was passed.

`EPERM` The door descriptor was not created by this process (with `door_create`(3DOOR)).

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Evolving
MT-Level	Safe

See Also `close`(2), `door_create`(3DOOR), `libdoor`(3LIB), `attributes`(5)

Name door_server_create – specify an alternative door server thread creation function

Synopsis

```
cc -mt [ flag ... ] file ... -ldoor  [ library ... ]
#include <door.h>

void (*) () door_server_create(void (*create_proc)(door_info_t*));
```

Description Normally, the doors library creates new door server threads in response to incoming concurrent door invocations automatically. There is no pre-defined upper limit on the number of server threads that the system creates in response to incoming invocations (1 server thread for each active door invocation). These threads are created with the default thread stack size and POSIX (see `standards(5)`) threads cancellation disabled. The created threads also have the `THR_BOUND` | `THR_DETACHED` attributes for Solaris threads and the `PTHREAD_SCOPE_SYSTEM` | `PTHREAD_CREATE_DETACHED` attributes for POSIX threads. The signal disposition, and scheduling class of the newly created thread are inherited from the calling thread (initially from the thread calling `door_create()`, and subsequently from the current active door server thread).

The `door_server_create()` function allows control over the creation of server threads needed for door invocations. The procedure *create_proc* is called every time the available server thread pool is depleted. In the case of private server pools associated with a door (see the `DOOR_PRIVATE` attribute in `door_create()`), information on which pool is depleted is passed to the create function in the form of a `door_info_t` structure. The `di_proc` and `di_data` members of the `door_info_t` structure can be used as a door identifier associated with the depleted pool. The *create_proc* procedure may limit the number of server threads created and may also create server threads with appropriate attributes (stack size, thread-specific data, POSIX thread cancellation, signal mask, scheduling attributes, and so forth) for use with door invocations.

The specified server creation function should create user level threads using `thr_create()` with the `THR_BOUND` flag, or in the case of POSIX threads, `pthread_create()` with the `PTHREAD_SCOPE_SYSTEM` attribute. The server threads make themselves available for incoming door invocations on this process by issuing a `door_return(NULL, 0, NULL, 0)`. In this case, the `door_return()` arguments are ignored. See `door_return(3DOOR)` and `thr_create(3C)`.

The server threads created by default are enabled for POSIX thread cancellations which may lead to unexpected thread terminations while holding resources (such as locks) if the client aborts the associated `door_call()`. See `door_call(3DOOR)`. Unless the server code is truly interested in notifications of client aborts during a door invocation and is prepared to handle such notifications using cancellation handlers, POSIX thread cancellation should be disabled for server threads using `pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL)`.

The *create_proc* procedure need not create any additional server threads if there is at least one server thread currently active in the process (perhaps handling another door invocation) or it may create as many as seen fit each time it is called. If there are no available server threads during an incoming door invocation, the associated `door_call()` blocks until a server thread becomes available. The *create_proc* procedure must be MT-Safe.

Return Values Upon successful completion, `door_server_create()` returns a pointer to the previous server creation function. This function has no failure mode (it cannot fail).

Examples **EXAMPLE 1** Creating door server threads.

The following example creates door server threads with cancellation disabled and an 8k stack instead of the default stack size:

```
#include <door.h>
#include <pthread.h>
#include <thread.h>

void *
my_thread(void *arg)
{
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
        door_return(NULL, 0, NULL, 0);
}
void
my_create(door_info_t *dip)
{
        thr_create(NULL, 8192, my_thread, NULL,
                   THR_BOUND | THR_DETACHED, NULL);
}
main( )
{
        (void)door_server_create(my_create);
         . . .
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Evolving
MT-Level	Safe

See Also door_bind(3DOOR), door_call(3DOOR), door_create(3DOOR), door_return(3DOOR), libdoor(3LIB), pthread_create(3C), pthread_setcancelstate(3C), thr_create(3C), attributes(5), cancellation(5), standards(5)

Name door_ucred – return credential information associated with the client

Synopsis

```
cc -mt [ flag ... ] file... -ldoor [ library... ]
#include <door.h>

int door_ucred(ucred_t **info);
```

Description The `door_ucred()` function returns credential information associated with the client, if any, of the current door invocation.

When successful, `door_ucred()` writes a pointer to a user credential to the location pointed to by *info* if that location was previously `NULL`. If that location was non-null, `door_ucred()` assumes that *info* points to a previously allocated `ucred_t` which is then reused. The location pointed to by *info* can be used multiple times before being freed. The value returned in *info* must be freed using `ucred_free`(3C).

The resulting user credential includes information about the effective user and group ID, the real user and group ID, all privilege sets and the calling PID.

The credential information associated with the client refers to the information from the immediate caller, not necessarily from the first thread in a chain of door calls.

Return Values Upon successful completion, `door_ucred()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error, in which case the memory location pointed to by the *info* argument is unchanged.

Errors The `door_ucred()` function will fail if:

`EAGAIN` The location pointed to by *info* was `NULL` and allocating memory sufficient to hold a ucred failed.

`EFAULT` The address of the *info* argument is invalid.

`EINVAL` There is no associated door client.

`ENOMEM` The location pointed to by *info* was `NULL` and allocating memory sufficient to hold a ucred failed.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving
MT-Level	Safe

See Also `door_call`(3DOOR), `door_create`(3DOOR), `ucred_get`(3C), `attributes`(5)

Name door_xcreate – create a door descriptor for a private door with per-door control over thread creation

Synopsis

```
#include <door.h>

typedef void door_server_procedure_t(void *, char *, size_t,
     door_desc_t *, uint_t);

typedef int door_xcreate_server_func_t(door_info_t *,
     void *(*)(void *), void *, void *);

typedef void door_xcreate_thrsetup_func_t(void *);

int door_xcreate(door_server_procedure_t *server_proceduere,
     void *cookie, uint_t attributes,
     door_xcreate_server_func_t *thr_create_func,
     door_xcreate_thrsetup_func_t *thr_setup_func, void *crcookie,
     int nthread);
```

Description The `door_xcreate()` function creates a private door to the given *server_procedure*, with per-door control over the creation of threads that will service invocations of that door. A private door is a door that has a private pool of threads that service calls to that door alone; non-private doors share a pool of service threads (see `door_create`(3DOOR)).

Creating private doors using `door_create()`

Prior to the introduction of `door_xcreate()`, a private door was created using `door_create()` specifying attributes including `DOOR_PRIVATE` after installing a suitable door server thread creation function using `door_server_create()`. During such a call to `door_create()`, the first server thread for that door is created by calling the door server function; you must therefore already have installed a custom door server creation function using `door_server_create()`. The custom server creation function is called at initial creation of a private door, and again whenever a new invocation uses the last available thread for that door. The function must decide whether it wants to increase the level of concurrency by creating an additional thread - if it decides not to then further invocations may have to wait for an existing active invocation to complete before they can proceed. Additional threads may be created using whatever thread attributes are desired in the application, and the application must specify a thread start function (to `thr_create`(3C) or `pthread_create`(3C)) which will perform a `door_bind()` to the newly-created door before calling `door_return`(`NULL`, 0, `NULL`, 0) to enter service. See `door_server_create`(3DOOR) and `door_bind`(3DOOR) for more information and for an example.

This “legacy” private door API is adequate for many uses, but has some limitations:

- The server thread creation function appointed via the `door_server_create()` is shared by all doors in the process. Private doors are distinguished from non-private in that the `door_info_t` pointer argument to the thread creation function is non-null for private doors; from the `door_info_t` the associated door server procedure is available via the `di_proc` member.
- If a library wishes to create a private door of which the application is essentially unaware it has no option but to inherit any function appointed with `door_server_create()` which may render the library door inoperable.
- Newly-created server threads must bind to the door they will service, but the door file descriptor to quote in `door_bind()` is not available in the `door_info_t` structure we receive a pointer to. The door file descriptor is returned as the result of `door_create()`, but the initial service thread is created during the call to `door_create()`. This leads to complexity in the startup of the service thread, and tends to force the use of global variables for the door file descriptors as per the example in `door_bind()`.

Creating private doors with `door_xcreate()`

The `door_xcreate()` function is purpose-designed for the creation of private doors and simplifies their use by moving responsibility for binding the new server thread and synchronizing with it into a library-provided thread startup function:

- The first three arguments to `door_xcreate()` are as you would use in `door_create()`: the door *server_procedure*, a private cookie to pass to that procedure whenever it is invoked for this door, and desired door attributes. The `DOOR_PRIVATE` attribute is implicit, and an additional attribute of `DOOR_NO_DEPLETION_CB` is available.
- Four additional arguments specify a server thread creation function to use for this door (must not be `NULL`), a thread setup function for new server threads (can be `NULL`), a cookie to pass to those functions, and the initial number of threads to create for this door.
- The `door_xcreate_server_func_t()` for creating server threads has differing semantics to those of a `door_server_func_t()` used in `door_server_create()`. In addition to a `door_info_t` pointer it also receives as arguments a library-provided thread start function and thread start argument that it must use, and the private cookie registered in the call to `door_xcreate()`. The nominated `door_xcreate_server_func_t()` must:
  - Return 0 if no additional thread is to be created, for example if it decides the current level of concurrency is sufficient. When the server thread creation function is invoked as part of a depletion callback (as opposed to during initial `door_xcreate()`) the `door_info_t` `di_attributes` member includes `DOOR_DEPLETION_CB`.
  - Otherwise attempt to create exactly one new thread using `thr_create()` or `pthread_create()`, with whatever thread attributes (stack size) are desired and quoting the implementation-provided thread start function and opaque data cookie. If the call to `thr_create()` or `pthread_create()` is successful then return 1, otherwise return -1.
  - Do not call `door_bind()` or request to enter service via `door_return`(`NULL`, 0, `NULL`, 0).

As in `door_server_create()` new server threads must be created `PTHREAD_SCOPE_SYSTEM` and `PTHREAD_CREATE_DETACHED` for POSIX threads, and `THR_BOUND` and `THR_DETACHED` for Solaris threads. The signal disposition and scheduling class of newly-created threads are inherited from the calling thread, initially from the thread calling `door_xcreate()` and subsequently from the current active door server thread.

- The library-provided thread start function performs the following operations in the order presented:
  - Calls the `door_xcreate_thrsetup_func_t()` if it is not `NULL`, passing the *crcookie*. You can use this setup function to perform custom service thread configuration that must be done from the context of the new thread. Typically this is to configure cancellation preferences, and possibly to associate application thread-specific-data with the newly-created server thread.

    If `thr_setup_func()` was `NULL` then a default is applied which will configure the new thread with `pthread_setcancelstate`(`PTHREAD_CANCEL_DISABLE`, `NULL`) and `pthread_setcanceltype`(`PTHREAD_CANCEL_DEFERRED`, `NULL`). If the server code is truly interested in notifications of client aborts during a door invocation then you will need to provide a `thr_setup_func()` that does not disable cancellations, and use `pthread_cleanup_push`(3C)and `pthread_cleanup_pop`(3C)as appropriate.
  - Binds the new thread to the door file descriptor using `door_bind()`.
  - Synchronizes with `door_xcreate()` so that the new server thread is known to have successfully completed `door_bind()` before `door_xcreate()` returns.
- The number of service threads to create at initial door creation time can be controlled through the *nthread* argument to `door_xcreate()`. The nominated `door_xcreate_server_func_t()` will be called *nthread* times. All *nthread* new server threads must be created successfully (`thr_create_func()` returns 1 for each) and all must succeed in binding to the new door; if fewer than *nthread* threads are created, or fewer than *nthread* succeed in binding, then `door_xcreate()` fails and any threads that were created are made to exit.

  No artificial maximum value is imposed on the *nthread* argument: it may be as high as system resources and available virtual memory permit. There is a small amount of additional stack usage in the `door_xcreate()` stack frame for each thread - up to 16 bytes in a 64-bit application. If there is unsufficient room to extend the stack for this purpose then `door_xcreate()` fails with `E2BIG`.

  The door attributes that can be selected in the call to `door_xcreate()` are the same as in `door_create()`, with `DOOR_PRIVATE` implied and `DOOR_NO_DEPLETION_CB` added:

  `DOOR_PRIVATE`
  : It is not necessary to include this attribute. The `door_xcreate()` interfaces only creates private doors.

DOOR_NO_DEPLETION_CB
Create the initial pool of *nthread* service threads, but do not perform further callbacks to the thr_create_func() for this door when the thread pool appears to be depleted at the start of a new door invocation. This allows you to select a fixed level of concurrency.

Another di_attribute is defined during thread depletion callbacks:

DOOR_DEPLETION_CB
This call to the server thread creation function is the result of a depletion callback. This attribute is not set when the function is called during initial door_xcreate().

The descriptor returned from door_xcreate() will be marked as close on exec (FD_CLOEXEC). Information about a door is available for all clients of a door using door_info(3DOOR). Applications concerned with security should not place secure information in door data that is accessible by door_info(). In particular, secure data should not be stored in the data item cookie.

A process can advertise a door in the file system name space using fattach(3C).

A door created with door_xcreate() may be revoked using door_revoke(3DOOR). This closes the associated file descriptor, and acts as a barrier to further door invocations, but existing active invocations are not guaranteed to have completed before door_revoke() returns. Server threads bound to a revoked door do not wakeup or exit automatically when the door is revoked.

Return Values Upon successful completion, door_xcreate() returns a non-negative value. Otherwise, door_xcreate() returns -1 and sets errno to indicate the error.

Errors The door_xcreate() function will fail if:

E2BIG The requested *nthread* is too large. A small amount of stack space is required for each thread we must start and synchronize with. If extending the door_xcreate() stack by the required amount will exceed the stack bounds then E2BIG is returned.

EBADF The attempt to door_bind() within the library-provided thread start function failed.

EINVAL Invalid attributes are passed, *nthread* is less than 1, or thr_create_func() is NULL. This is also returned if thr_create_func() returns 0 (no thread creation attempted) during door_xcreate().

EMFILE The process has too many open descriptors.

ENOMEM Insufficient memory condition while creating the door.

ENOTSUP A door_xcreate() call was attempted from a fork handler.

EPIPE A call to the nominated thr_create_func() returned -1 indicating that pthread_create() or thr_create() failed.

Examples **EXAMPLE 1** Create a private door with an initial pool of 10 server threads

Create a private door with an initial pool of 10 server threads. Threads are created with the minimum required attributes and there is no thread setup function. Use `fattach()` to advertise the door in the filesystem namespace.

```
static pthread_attr_t tattr;

/*
 * Simplest possible door_xcreate_server_func_t.  Always attempt to
 * create a thread, using the previously initialized attributes for
 * all threads. We must use the start function and argument provided,
 * and make no use of our private mycookie argument.
 */
int
thrcreatefunc(door_info_t *dip, void *(*startf)(void *),
    void *startfarg, void *mycookie)
{
        if (pthread_create(NULL, &tattr, startf, startfarg) != 0) {
                perror("thrcreatefunc: pthread_create");
                return (-1);
        }

        return (1);
}

/*
 * Dummy door server procedure - does no processing.
 */
void
door_proc(void *cookie, char *argp, size_t argsz, door_desc_t *descp,
    uint_t n)
{
    door_return (NULL, 0, NULL, 0);
}

int
main(int argc, char *argv[])
{
    struct stat buf;
    int did;

    /*
     * Setup thread attributes - minimum required.
     */
    (void) pthread_attr_init(&tattr);
    (void) pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);
    (void) pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
```

EXAMPLE 1 Create a private door with an initial pool of 10 server threads *(Continued)*

```
    /*
     * Create a private door with an initial pool of 10 server threads.
     */
    did = door_xcreate(door_proc, NULL, 0, thrcreatefunc, NULL, NULL,
                       10);

    if (did == -1) {
        perror("door_xcreate");
        exit(1);
    }

    if (stat(DOORPATH, &buf) < 0) {
        int newfd;

        if ((newfd = creat(DOORPATH, 0644)) < 0) {
            perror("creat");
            exit(1);
        }
        (void) close(newfd);
    }

    (void) fdetach(DOORPATH);

    (void) fdetach(DOORPATH);
    if (fattach(did, DOORPATH) < 0) {
        perror("fattach");
        exit(1);
    }

    (void) fprintf(stderr, "Pausing in main\n");
    (void) pause();
}
```

EXAMPLE 2 Create a private door with exactly one server thread and no callbacks for additional threads

Create a private door with exactly one server thread and no callbacks for additional threads. Use a server thread stacksize of 32K, and specify a thread setup function.

```
#define DOORPATH        "/tmp/grmdoor"

static pthread_attr_t tattr;

/*
 * Thread setup function - configuration that must be performed from
```

EXAMPLE 2 Create a private door with exactly one server thread and no callbacks for additional threads *(Continued)*

```
 * the conext of the new thread.  The mycookie argument is the
 * second-to-last argument from door_xcreate.
 */
void
thrsetupfunc(void *mycookie)
{
    /*
     * If a thread setup function is specified it must do the
     * following at minimum.
     */
    (void) pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

    /*
     * The default thread setup functions also performs the following
     * to disable thread cancellation notifications, so that server
     * threads are not cancelled when a client aborts a door call.
     * This is not a requirement.
     */
    (void) pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);

    /*
     * Now we can go on to perform other thread initialization,
     * for example to allocate and initialize some thread-specific data
     * for this thread;  for thread-specific data you can use a
       destructor function in pthread_key_create if you want to perform
       any actions if/when a door server thread exits.
     */
}

/*
 * The door_xcreate_server_func_t we will use for server thread
 * creation. The mycookie argument is the second-to-last argument
 * from door_xcreate.
 */
int
thrcreatefunc(door_info_t *dip, void *(*startf)(void *),
    void *startfarg, void *mycookie)
{
    if (pthread_create(NULL, &tattr, startf, startfarg) != 0) {
            perror("thrcreatefunc: pthread_create");
            return (-1);
    }

    return (1);
```

EXAMPLE 2 Create a private door with exactly one server thread and no callbacks for additional threads *(Continued)*

```
}

/*
 * Door procedure.  The cookie received here is the second arg to
 * door_xcreate.
 */
void
door_proc(void *cookie, char *argp, size_t argsz, door_desc_t *descp,
    uint_t n)
{
    (void) door_return(NULL, 0, NULL, 0);
}

int
main(int argc, char *argv[])
{
    struct stat buf;
    int did;

    /*
     * Configure thread attributes we will use in thrcreatefunc.
     * The PTHREAD_CREATE_DETACHED and PTHREAD_SCOPE_SYSTEM are
     * required.
     */
    (void) pthread_attr_init(&tattr);
    (void) pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);
    (void) pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
    (void) pthread_attr_setstacksize(&tattr, 16 * 1024);

    /*
     * Create a private door with just one server thread and asking for
         * no further callbacks on thread pool depletion during an
         * invocation.
         */
    did = door_xcreate(door_proc, NULL, DOOR_NO_DEPLETION_CB,
        thrcreatefunc, thrsetupfunc, NULL, 1);

    if (did == -1) {
            perror("door_xcreate");
            exit(1);
    }

    if (stat(DOORPATH, &buf) < 0) {
            int newfd;
```

EXAMPLE 2 Create a private door with exactly one server thread and no callbacks for additional threads *(Continued)*

```
                if ((newfd = creat(DOORPATH, 0644)) < 0) {
                        perror("creat");
                        exit(1);
                }
                (void) close(newfd);
        }

        (void) fdetach(DOORPATH);
        if (fattach(did, DOORPATH) < 0) {
                perror("fattach");
                exit(1);
        }

        (void) fprintf(stderr, "Pausing in main\n");
        (void) pause();
}
```

Attributes

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Architecture	all
Availability	SUNWcsu
Interface Stability	Committed
MT-Level	Safe

See Also

door_bind(3DOOR), door_call(3DOOR), door_create(3DOOR), door_info(3DOOR), door_revoke(3DOOR), door_server_create(3DOOR), fattach(3C), libdoor(3LIB), pthread_create(3C), pthread_cleanup_pop(3C), pthread_cleanup_push(3C), thr_create(3C), attributes(5), cancellation(5)

Name fdatasync – synchronize a file's data

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <unistd.h>

int fdatasync(int fildes);
```

Description The `fdatasync()` function forces all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state.

The functionality is as described for `fsync`(3C) (with the symbol `_XOPEN_REALTIME` defined), with the exception that all I/O operations are completed as defined for synchronised I/O data integrity completion.

Return Values If successful, the `fdatasync()` function returns `0`. Otherwise, the function returns `−1` and sets `errno` to indicate the error. If the `fdatasync()` function fails, outstanding I/O operations are not guaranteed to have been completed.

Errors The `fdatasync()` function will fail if:

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

`EINVAL` The system does not support synchronized I/O for this file.

`ENOSYS` The function `fdatasync()` is not supported by the system.

In the event that any of the queued I/O operations fail, `fdatasync()` returns the error conditions defined for `read`(2) and `write`(2).

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also `fcntl`(2), `open`(2), `read`(2), `write`(2), `fsync`(3C), `aio_fsync`(3RT), `fcntl.h`(3HEAD), `attributes`(5), `standards`(5)

Name lio_listio – list directed I/O

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[], int nent,
     struct sigevent *restrict sig);
```

Description The `lio_listio()` function allows the calling process, LWP, or thread, to initiate a list of I/O requests within a single function call.

The *mode* argument takes one of the values `LIO_WAIT` or `LIO_NOWAIT` declared in `<aio.h>` and determines whether the function returns when the I/O operations have been completed, or as soon as the operations have been queued. If the *mode* argument is `LIO_WAIT`, the function waits until all I/O is complete and the *sig* argument is ignored.

If the *mode* argument is `LIO_NOWAIT`, the function returns immediately, and asynchronous notification occurs, according to the *sig* argument, when all the I/O operations complete. If *sig* is `NULL`, or the `sigev_signo` member of the `sigevent` structure referenced by *sig* is zero, then no asynchronous notification occurs. If *sig* is not `NULL`, asynchronous notification occurs when all the requests in *list* have completed. If *sig*–>`sigev_notify` is `SIGEV_NONE`, then no signal will be posted upon I/O completion, but the error status and the return status for the operation will be set appropriately. If *sig*–>`sigev_notify` is `SIGEV_SIGNAL`, then the signal specified in *sig*–>`sigev_signo` will be sent to the process. If the `SA_SIGINFO` flag is set for that signal number, then the signal will be queued to the process and the value specified in *sig*–>`sigev_value` will be the `si_value` component of the generated signal (see `siginfo.h`(3HEAD)). If *sig*–>`sigev_notify` is `SIGEV_PORT`, then upon I/O completion an event notification will be sent to the event port determined in the `port_notify_t` structure addressed by the `sival_ptr` (see `signal.h`(3HEAD)).

The I/O requests enumerated by *list* are submitted in an unspecified order.

The *list* argument is an array of pointers to `aiocb` structures. The array contains *nent* elements. The array may contain null elements, which are ignored.

The *aio_lio_opcode* field of each `aiocb` structure specifies the operation to be performed. The supported operations are `LIO_READ`, `LIO_WRITE`, and `LIO_NOP`; these symbols are defined in `<aio.h>`. The `LIO_NOP` operation causes the list entry to be ignored. If the *aio_lio_opcode* element is equal to `LIO_READ,` then an I/O operation is submitted as if by a call to `aio_read`(3RT) with the *aiocbp* equal to the address of the `aiocb` structure. If the *aio_lio_opcode* element is equal to `LIO_WRITE`, then an I/O operation is submitted as if by a call to `aio_write`(3RT) with the *aiocbp* equal to the address of the `aiocb` structure.

The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

The *aio_buf* member specifies the address of the buffer to or from which the data is to be transferred.

The *aio_nbytes* member specifies the number of bytes of data to be transferred.

The members of the *aiocb* structure further describe the I/O operation to be performed, in a manner identical to that of the corresponding `aiocb` structure when used by the `aio_read`(3RT) and `aio_write`(3RT) functions.

The *nent* argument specifies how many elements are members of the list, that is, the length of the array.

The behavior of this function is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with `aio_fildes`. (see `fcntl.h`(3HEAD) definitions of `O_DSYNC` and `O_SYNC`.)

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *aiocbp*->`aio_fildes`.

Return Values If the *mode* argument has the value `LIO_NOWAIT`, and the I/O operations are successfully queued, `lio_listio()` returns `0`; otherwise, it returns −`1`, and sets `errno` to indicate the error.

If the *mode* argument has the value `LIO_WAIT`, and all the indicated I/O has completed successfully, `lio_listio()` returns `0`; otherwise, it returns −`1`, and sets `errno` to indicate the error.

In either case, the return value only indicates the success or failure of the `lio_listio()` call itself, not the status of the individual I/O requests. In some cases, one or more of the I/O requests contained in the list may fail. Failure of an individual request does not prevent completion of any other individual request. To determine the outcome of each I/O request, the application must examine the error status associated with each *aiocb* control block. Each error status so returned is identical to that returned as a result of an `aio_read`(3RT) or `aio_write`(3RT) function.

Errors The `lio_listio()` function will fail if:

`EAGAIN` The resources necessary to queue all the I/O requests were not available. The error status for each request is recorded in the `aio_error` member of the corresponding `aiocb` structure, and can be retrieved using `aio_error`(3RT).

`EAGAIN` The number of entries indicated by *nent* would cause the system-wide limit `AIO_MAX` to be exceeded.

`EINVAL` The *mode* argument is an improper value, or the value of *nent* is greater than `AIO_LISTIO_MAX`.

`EINTR` A signal was delivered while waiting for all I/O requests to complete during an `LIO_WAIT` operation. Note that, since each I/O operation invoked by `lio_listio()` may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited. Outstanding I/O requests are not canceled, and the application can use `aio_fsync(3RT)` to determine if any request was initiated; `aio_return(3RT)` to determine if any request has completed; or `aio_error(3RT)` to determine if any request was canceled.

`EIO` One or more of the individual I/O operations failed. The application can use `aio_error(3RT)` to check the error status for each `aiocb` structure to determine the individual request(s) that failed.

`ENOSYS` The `lio_listio()` function is not supported by the system.

In addition to the errors returned by the `lio_listio()` function, if the `lio_listio()` function succeeds or fails with errors of `EAGAIN`, `EINTR`, or `EIO`, then some of the I/O specified by the list may have been initiated. If the `lio_listio()` function fails with an error code other than `EAGAIN`, `EINTR`, or `EIO`, no operations from the list have been initiated. The I/O operation indicated by each list element can encounter errors specific to the individual read or write function being performed. In this event, the error status for each `aiocb` control block contains the associated error code. The error codes that can be set are the same as would be set by a `read(2)` or `write(2)` function, with the following additional error codes possible:

`EAGAIN` The requested I/O operation was not queued due to resource limitations.

`ECANCELED` The requested I/O was canceled before the I/O completed due to an explicit `aio_cancel(3RT)` request.

`EFBIG` The *aiocbp*->`aio_lio_opcode` is `LIO_WRITE`, the file is a regular file, *aiocbp*->`aio_nbytes` is greater than 0, and the *aiocbp*->`aio_offset` is greater than or equal to the offset maximum in the open file description associated with *aiocbp*->`aio_fildes.`

`EINPROGRESS` The requested I/O is in progress.

`EOVERFLOW` The *aiocbp*->`aio_lio_opcode` is `LIO_READ`, the file is a regular file, *aiocbp*->`aio_nbytes` is greater than 0, and the *aiocbp*->`aio_offset` is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with *aiocbp*->`aio_fildes.`

Usage

The `lio_listio()` function has a transitional interface for 64-bit file offsets. See `lf64(5)`.

Attributes

See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also close(2), exec(2), exit(2), fork(2), lseek(2), read(2), write(2), aio_cancel(3RT), aio_error(3RT), aio_fsync(3RT), aio_read(3RT), aio_return(3RT), aio_write(3RT), aio.h(3HEAD), fcntl.h(3HEAD), siginfo.h(3HEAD), signal.h(3HEAD), attributes(5), lf64(5), standards(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name mq_close – close a message queue

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Description The `mq_close()` function removes the association between the message queue descriptor, *mqdes*, and its message queue. The results of using this message queue descriptor after successful return from this `mq_close()`, and until the return of this message queue descriptor from a subsequent `mq_open(3RT)`, are undefined.

If the process (or thread) has successfully attached a notification request to the message queue via this *mqdes*, this attachment is removed and the message queue is available for another process to attach for notification.

Return Values Upon successful completion, `mq_close()` returns `0`; otherwise, the function returns `−1` and sets `errno` to indicate the error condition.

Errors The `mq_close()` function will fail if:

`EBADF` The *mqdes* argument is an invalid message queue descriptor.

`ENOSYS` The `mq_open()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `mqueue.h(3HEAD)`, `mq_notify(3RT)`, `mq_open(3RT)`, `mq_unlink(3RT)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name mq_getattr – get message queue attributes

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
```

Description The *mqdes* argument specifies a message queue descriptor. The `mq_getattr()` function is used to get status information and attributes of the message queue and the open message queue description associated with the message queue descriptor. The results are returned in the *mq_attr* structure referenced by the *mqstat* argument.

Upon return, the following members will have the values associated with the open message queue description as set when the message queue was opened and as modified by subsequent `mq_setattr(3RT)` calls:

`mq_flags` message queue flags

The following attributes of the message queue are returned as set at message queue creation:

`mq_maxmsg` maximum number of messages

`mq_msgsize` maximum message size

`mq_curmsgs` number of messages currently on the queue.

Return Values Upon successful completion, the `mq_getattr()` function returns `0`. Otherwise, the function returns −1 and sets `errno` to indicate the error.

Errors The `mq_getattr()` function will fail if:

`EBADF` The *mqdes* argument is not a valid message queue descriptor.

`ENOSYS` The `mq_getattr()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `msgctl(2)`, `msgget(2)`, `msgrcv(2)`, `msgsnd(2)`, `mqueue.h(3HEAD)`, `mq_open(3RT)`, `mq_send(3RT)`, `mq_setattr(3RT)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name mq_notify – notify process (or thread) that a message is available on a queue

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

Description The `mq_notify()` function provides an asynchronous mechanism for processes to receive notice that messages are available in a message queue, rather than synchronously blocking (waiting) in `mq_receive`(3RT).

If *notification* is not `NULL`, this function registers the calling process to be notified of message arrival at an empty message queue associated with the message queue descriptor, *mqdes*. The notification specified by *notification* will be sent to the process when the message queue transitions from empty to non-empty. At any time, only one process may be registered for notification by a specific message queue. If the calling process or any other process has already registered for notification of message arrival at the specified message queue, subsequent attempts to register for that message queue will fail.

The *notification* argument points to a structure that defines both the signal to be generated and how the calling process will be notified upon I/O completion. If *notification*->`sigev_notify` is `SIGEV_NONE`, then no signal will be posted upon I/O completion, but the error status and the return status for the operation will be set appropriately. If *notification*->`sigev_notify` is `SIGEV_SIGNAL`, then the signal specified in *notification*->`sigev_signo` will be sent to the process. If the `SA_SIGINFO` flag is set for that signal number, then the signal will be queued to the process and the value specified in *notification*->`sigev_value` will be the `si_value` component of the generated signal (see `siginfo.h`(3HEAD)).

If *notification* is `NULL` and the process is currently registered for notification by the specified message queue, the existing registration is removed. The message queue is then available for future registration.

When the notification is sent to the registered process, its registration is removed. The message queue is then be available for registration.

If a process has registered for notification of message arrival at a message queue and some processes is blocked in `mq_receive`(3RT) waiting to receive a message when a message arrives at the queue, the arriving message will be received by the appropriate `mq_receive`(3RT), and no notification will be sent to the registered process. The resulting behavior is as if the message queue remains empty, and this notification will not be sent until the next arrival of a message at this queue.

Any notification registration is removed if the calling process either closes the message queue or exits.

Return Values Upon successful completion, `mq_notify()` returns `0`; otherwise, it returns `−1` and sets `errno` to indicate the error.

Errors The `mq_notify()` function will fail if:

`EBADF` The *mqdes* argument is not a valid message queue descriptor.

`EBUSY` A process is already registered for notification by the message queue.

`ENOSYS` The `mq_notify()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `mqueue.h(3HEAD)`, `siginfo.h(3HEAD)`, `signal.h(3HEAD)`, `mq_close(3RT)`, `mq_open(3RT)`, `mq_receive(3RT)`, `mq_send(3RT)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name mq_open – open a message queue

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag,
     /* unsigned long mode, mq_attr attr */  ...);
```

Description The `mq_open()` function establishes the connection between a process and a message queue with a message queue descriptor. It creates a open message queue description that refers to the message queue, and a message queue descriptor that refers to that open message queue description. The message queue descriptor is used by other functions to refer to that message queue.

The *name* argument points to a string naming a message queue. The *name* argument must conform to the construction rules for a path-name. If *name* is not the name of an existing message queue and its creation is not requested, `mq_open()` fails and returns an error. The first character of *name* must be a slash (`/`) character and the remaining characters of *name* cannot include any slash characters. For maximum portability, *name* should include no more than 14 characters, but this limit is not enforced.

The *oflag* argument requests the desired receive and/or send access to the message queue. The requested access permission to receive messages or send messages is granted if the calling process would be granted read or write access, respectively, to a file with the equivalent permissions.

The value of *oflag* is the bitwise inclusive OR of values from the following list. Applications must specify exactly one of the first three values (access modes) below in the value of *oflag*:

`O_RDONLY` Open the message queue for receiving messages. The process can use the returned message queue descriptor with `mq_receive(3RT)`, but not `mq_send(3RT)`. A message queue may be open multiple times in the same or different processes for receiving messages.

`O_WRONLY` Open the queue for sending messages. The process can use the returned message queue descriptor with `mq_send(3RT)` but not `mq_receive(3RT)`. A message queue may be open multiple times in the same or different processes for sending messages.

`O_RDWR` Open the queue for both receiving and sending messages. The process can use any of the functions allowed for `O_RDONLY` and `O_WRONLY`. A message queue may be open multiple times in the same or different processes for sending messages.

Any combination of the remaining flags may additionally be specified in the value of *oflag*:

`O_CREAT` This option is used to create a message queue, and it requires two additional arguments: *mode*, which is of type `mode_t`, and *attr*, which is pointer to a `mq_attr` structure. If the pathname, *name*, has already been used to create a message queue that still exists, then this flag has no effect, except as noted under O_EXCL (see below). Otherwise, a message queue is created without any messages in it.

The user ID of the message queue is set to the effective user ID of process, and the group ID of the message queue is set to the effective group ID of the process. The file permission bits are set to the value of *mode*, and modified by clearing all bits set in the file mode creation mask of the process (see umask(2)).

If *attr* is non-NULL and the calling process has the appropriate privilege on *name*, the message queue *mq_maxmsg* and *mq_msgsize* attributes are set to the values of the corresponding members in the mq_attr structure referred to by *attr*. If *attr* is non-NULL, but the calling process does not have the appropriate privilege on *name*, the mq_open() function fails and returns an error without creating the message queue.

O_EXCL — If both O_EXCL and O_CREAT are set, mq_open() will fail if the message queue *name* exists. The check for the existence of the message queue and the creation of the message queue if it does not exist are atomic with respect to other processes executing mq_open() naming the same *name* with both O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are not set, the result is undefined.

O_NONBLOCK — The setting of this flag is associated with the open message queue description and determines whether a mq_send(3RT) or mq_receive(3RT) waits for resources or messages that are not currently available, or fails with errno set to EAGAIN. See mq_send(3RT) and mq_receive(3RT) for details.

Return Values Upon successful completion, mq_open() returns a message queue descriptor; otherwise the function returns (mqd_t)−1 and sets errno to indicate the error condition.

Errors The mq_open() function will fail if:

EACCES — The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the message queue is denied.

EEXIST — O_CREAT and O_EXCL are set and the named message queue already exists.

EINTR — The mq_open() operation was interrupted by a signal.

EINVAL — The mq_open() operation is not supported for the given name, or O_CREAT was specified in *oflag*, the value of *attr* is not NULL, and either mq_maxmsg or mq_msgsize was less than or equal to zero.

EMFILE — The number of open message queue descriptors in this process exceeds MQ_OPEN_MAX, of the number of open file descriptors in this process exceeds OPEN_MAX.

ENAMETOOLONG — The length of the *name* string exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.

ENFILE — Too many message queues are currently open in the system.

ENOENT — O_CREAT is not set and the named message queue does not exist.

ENOSPC — There is insufficient space for the creation of the new message queue.

ENOSYS — The mq_open() function is not supported by the system.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also exec(2), exit(2), umask(2), sysconf(3C), mqueue.h(3HEAD), mq_close(3RT), mq_receive(3RT), mq_send(3RT), mq_setattr(3RT), mq_unlink(3RT), attributes(5), standards(5)

Notes Due to the manner in which message queues are implemented, they should not be considered secure and should not be used in security-sensitive applications.

Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name mq_receive, mq_timedreceive, mq_reltimedreceive_np – receive a message from a message queue

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
     unsigned *msg_prio);

#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
     size_t msg_len, unsigned *restrict msg_prio,
     const struct timespec *restrict abs_timeout);

ssize_t mq_reltimedreceive_np(mqd_t mqdes,
     char *restrict msg_ptr, size_t msg_len,
     unsigned *restrict msg_prio,
     const struct timespec *restrict rel_timeout);
```

Description The `mq_receive()` function receives the oldest of the highest priority message(s) from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by *msg_len*, is less than the `mq_msgsize` member of the message queue, the function fails and returns an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by *msg_ptr*.

If the value of *msg_len* is greater than {`SSIZE_MAX`}, the result is implementation-defined.

If *msg_prio* is not `NULL`, the priority of the selected message is stored in the location referenced by *msg_prio*.

If the specified message queue is empty and `O_NONBLOCK` is not set in the message queue description associated with *mqdes*, (see `mq_open`(3RT) and `mq_setattr`(3RT)), `mq_receive()` blocks, waiting until a message is enqueued on the message queue, or until `mq_receive()` is interrupted by a signal. If more than one process (or thread) is waiting to receive a message when a message arrives at an empty queue, then the process of highest priority that has been waiting the longest is selected to receive the message. If the specified message queue is empty and `O_NONBLOCK` is set in the message queue description associated with *mqdes*, no message is removed from the queue, and `mq_receive()` returns an error.

The `mq_timedreceive()` function receives the oldest of the highest priority messages from the message queue specified by *mqdes* as described for the `mq_receive()` function. However, if `O_NONBLOCK` was not specified when the message queue was opened with the `mq_open`(3RT) function, and no message exists on the queue to satisfy the receive, the wait for such a message is terminated when the specified timeout expires. If `O_NONBLOCK` is set, this function is equivalent to `mq_receive()`.

The `mq_reltimedreceive_np()` function is identical to the `mq_timedreceive()` function, except that the timeout is specified as a relative time interval.

For `mq_timedreceive()`, the timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` clock (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

For `mq_reltimedreceive_np()`, the timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` argument is defined in the `<time.h>` header.

Under no circumstance does the operation fail with a timeout if a message can be removed from the message queue immediately. The validity of the timeout parameter need not be checked if a message can be removed from the message queue immediately.

Return Values Upon successful completion, `mq_receive()`, `mq_timedreceive()`, and `mq_reltimedreceive_np()` return the length of the selected message in bytes and the message is removed from the queue. Otherwise, no message is removed from the queue, the functions return a value of `−1`, and sets `errno` to indicate the error condition.

Errors The `mq_receive()`, `mq_timedreceive()`, and `mq_reltimedreceive_np()` functions will fail if:

`EAGAIN` `O_NONBLOCK` was set in the message description associated with *mqdes*, and the specified message queue is empty.

`EBADF` The *mqdes* argument is not a valid message queue descriptor open for reading.

`EINTR` The function was interrupted by a signal.

`EINVAL` The process or thread would have blocked, and the timeout parameter specified a nanoseconds field value less than zero or greater than or equal to 1,000 million.

`EMSGSIZE` The specified message buffer size, *msg_len*, is less than the message size member of the message queue.

`ETIMEDOUT` The `O_NONBLOCK` flag was not set when the message queue was opened, but no message arrived on the queue before the specified timeout expired.

The `mq_receive()`, `mq_timedreceive()`, and `mq_reltimedreceive_np()` functions may fail if:

`EBADMSG` A data corruption problem with the message has been detected.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	MT-Safe

The mq_receive() and mq_timedreceive() functions are Standard. The mq_reltimedreceive_np() function is Stable.

See Also mqueue.h(3HEAD), mq_open(3RT), mq_send(3RT), mq_setattr(3RT), attributes(5), standards(5)

Name mq_send, mq_timedsend, mq_reltimedsend_np – send a message to a message queue

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
     unsigned msg_prio);

#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
     size_t msg_len, unsigned msg_prio,
     const struct timespec *restrict abs_timeout);

int mq_reltimedsend_np(mqd_t mqdes, const char *msg_ptr,
     size_t msg_len, unsigned msg_prio,
     const struct timespec *restrict rel_timeout);
```

Description The `mq_send()` function adds the message pointed to by the argument *msg_ptr* to the message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message in bytes pointed to by *msg_ptr*. The value of *msg_len* is less than or equal to the *mq_msgsize* attribute of the message queue, or `mq_send()` fails.

If the specified message queue is not full, `mq_send()` behaves as if the message is inserted into the message queue at the position indicated by the *msg_prio* argument. A message with a larger numeric value of *msg_prio* is inserted before messages with lower values of *msg_prio*. A message will be inserted after other messages in the queue, if any, with equal *msg_prio*. The value of *msg_prio* must be greater than zero and less than or equal to `MQ_PRIO_MAX`.

If the specified message queue is full and `O_NONBLOCK` is not set in the message queue description associated with *mqdes* (see `mq_open(3RT)` and `mq_setattr(3RT)`), `mq_send()` blocks until space becomes available to enqueue the message, or until `mq_send()` is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue, then the thread of the highest priority which has been waiting the longest is unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and `O_NONBLOCK` is set in the message queue description associated with *mqdes*, the message is not queued and `mq_send()` returns an error.

The `mq_timedsend()` function adds a message to the message queue specified by *mqdes* in the manner defined for the `mq_send()` function. However, if the specified message queue is full and `O_NONBLOCK` is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue is terminated when the specified timeout expires. If `O_NONBLOCK` is set in the message queue description, this function is equivalent to `mq_send()`.

The `mq_reltimedsend_np()` function is identical to the `mq_timedsend()` function, except that the timeout is specified as a relative time interval.

For `mq_timedsend()`, the timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` clock (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

For `mq_reltimedsend_np()`, the timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` argument is defined in the `<time.h>` header.

Under no circumstance does the operation fail with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the timeout parameter need not be checked when there is sufficient room in the queue.

Return Values Upon successful completion, `mq_send()`, `mq_timedsend()`, and `mq_reltimedsend_np()` return `0`. Otherwise, no message is enqueued, the functions return −1, and `errno` is set to indicate the error.

Errors The `mq_send()`, `mq_timedsend()`, and `mq_reltimedsend_np()` functions will fail if:

`EAGAIN` The `O_NONBLOCK` flag is set in the message queue description associated with *mqdes*, and the specified message queue is full.

`EBADF` The *mqdes* argument is not a valid message queue descriptor open for writing.

`EINTR` A signal interrupted the function call.

`EINVAL` The value of *msg_prio* was outside the valid range.

`EINVAL` The process or thread would have blocked, and the timeout parameter specified a nanoseconds field value less than zero or greater than or equal to 1,000 million.

`EMSGSIZE` The specified message length, *msg_len*, exceeds the message size attribute of the message queue.

`ETIMEDOUT` The `O_NONBLOCK` flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	MT-Safe

The mq_send() and mq_timedsend() functions are Standard. The mq_reltimedsend_np() function is Stable.

See Also sysconf(3C), mqueue.h(3HEAD), mq_open(3RT), mq_receive(3RT), mq_setattr(3RT), attributes(5), standards(5)

Name mq_setattr – set/get message queue attributes

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat,
     struct mq_attr *omqstat);
```

Description The `mq_setattr()` function is used to set attributes associated with the open message queue description referenced by the message queue descriptor specified by *mqdes*.

The message queue attributes corresponding to the following members defined in the `mq_attr` structure are set to the specified values upon successful completion of `mq_setattr()`:

`mq_flags` The value of this member is either `0` or `O_NONBLOCK`.

The values of `mq_maxmsg`, `mq_msgsize`, and `mq_curmsgs` are ignored by `mq_setattr()`.

If *omqstat* is non-`NULL`, `mq_setattr()` stores, in the location referenced by *omqstat*, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to `mq_getattr()` at that point.

Return Values Upon successful completion, `mq_setattr()` returns `0` and the attributes of the message queue will have been changed as specified. Otherwise, the message queue attributes are unchanged, and the function returns −`1` and sets `errno` to indicate the error.

Errors The `mq_setattr()` function will fail if:

`EBADF` The *mqdes* argument is not a valid message queue descriptor.

`ENOSYS` The `mq_setattr()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `msgctl(2)`, `msgget(2)`, `msgrcv(2)`, `msgsnd(2)`, `mq_getattr(3RT)`, `mq_open(3RT)`, `mq_receive(3RT)`, `mq_send(3RT)`, `mqueue.h(3HEAD)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −`1` and set `errno` to `ENOSYS`.

Name mq_unlink – remove a message queue

Synopsis
```
cc [ flag... ] file... -lrt [ library... ]
#include <mqueue.h>

int mq_unlink(const char *name);
```

Description The `mq_unlink()` function removes the message queue named by the pathname *name*. After a successful call to `mq_unlink()` with *name*, a call to `mq_open(3RT)` with *name* fails if the flag `O_CREAT` is not set in *flags*. If one or more processes have the message queue open when `mq_unlink()` is called, destruction of the message queue is postponed until all references to the message queue have been closed. Calls to `mq_open(3RT)` to re-create the message queue may fail until the message queue is actually removed. However, the `mq_unlink()` call need not block until all references have been closed; it may return immediately.

Return Values Upon successful completion, `mq_unlink()` returns `0`; otherwise, the named message queue is not changed by this function call, the function returns `−1` and sets `errno` to indicate the error.

Errors The `mq_unlink()` function will fail if:

`EACCES` Permission is denied to unlink the named message queue.

`ENAMETOOLONG` The length of the *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

`ENOENT` The named message queue, *name*, does not exist.

`ENOSYS` `mq_unlink()` is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `mqueue.h(3HEAD)`, `mq_close(3RT)`, `mq_open(3RT)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name nanosleep – high resolution sleep

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <time.h>

int nanosleep(const struct timespec *rqtp,
     struct timespec *rmtp);
```

Description The `nanosleep()` function causes the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time will not be less than the time specified by *rqtp*, as measured by the system clock, `CLOCK_REALTIME`.

The use of the `nanosleep()` function has no effect on the action or blockage of any signal.

Return Values If the `nanosleep()` function returns because the requested time has elapsed, its return value is `0`.

If the `nanosleep()` function returns because it has been interrupted by a signal, the function returns a value of −1 and sets `errno` to indicate the interruption. If the *rmtp* argument is non-`NULL`, the `timespec` structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is `NULL`, the remaining time is not returned.

If `nanosleep()` fails, it returns −1 and sets `errno` to indicate the error.

Errors The `nanosleep()` function will fail if:

`EINTR` The `nanosleep()` function was interrupted by a signal.

`EINVAL` The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

`ENOSYS` The `nanosleep()` function is not supported by this implementation.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also sleep(3C), time.h(3HEAD), attributes(5), standards(5)

Name proc_service – process service interfaces

Synopsis

```
#include <proc_service.h>

ps_err_e ps_pdmodel(struct ps_prochandle *ph,
     int *data_model);

ps_err_e ps_pglobal_lookup(struct ps_prochandle *ph,
     const char *object_name, const char *sym_name,
     psaddr_t *sym_addr);

ps_err_e ps_pglobal_sym(struct ps_prochandle *ph,
     const char *object_name, const char *sym_name,
     ps_sym_t *sym);

ps_err_e ps_pread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_pwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_pdread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_pdwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_ptread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_ptwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_pstop(struct ps_prochandle *ph);

ps_err_e ps_pcontinue(struct ps_prochandle *ph);

ps_err_e ps_lstop(struct ps_prochandle *ph, lwpid_t lwpid);

ps_err_e ps_lcontinue(struct ps_prochandle *ph, lwpid_t lwpid);

ps_err_e ps_lgetregs(struct ps_prochandle *ph, lwpid_t lwpid,
     prgregset_t gregset);

ps_err_e ps_lsetregs(struct ps_prochandle *ph, lwpid_t lwpid,
     const prgregset_t gregset);

ps_err_e ps_lgetfpregs(struct ps_prochandle *ph, lwpid_t lwpid,
     prfpregset_t *fpregset);

ps_err_e ps_lsetfpregs(struct ps_prochandle *ph, lwpid_t lwpid,
     const prfpregset_t *fpregset);

ps_err_e ps_pauxv(struct ps_prochandle *ph,
     const auxv_t **auxp);

ps_err_e ps_kill(struct ps_prochandle *ph, int sig);
```

```
ps_err_e ps_lrolltoaddr(struct ps_prochandle *ph,
     lwpid_t lwpid, psaddr_t go_addr, psaddr_t stop_addr);

void ps_plog(const char *fmt);
```

SPARC
```
ps_err_e ps_lgetxregsize(struct ps_prochandle *ph,
     lwpid_t lwpid, int *xregsize);

ps_err_e ps_lgetxregs(struct ps_prochandle *ph,
     lwpid_t lwpid, caddr_t xregset);

ps_err_e ps_lsetxregs(struct ps_prochandle *ph,
     lwpid_t lwpid, caddr_t xregset);
```

x86
```
ps_err_e ps_lgetLDT(struct ps_prochandle *ph, lwpid_t lwpid,
     struct ssd *ldt);
```

Description

Every program that links `libthread_db` or `librtld_db` must provide a set of process control primitives that allow `libthread_db` and `librtld_db` to access memory and registers in the target process, to start and to stop the target process, and to look up symbols in the target process. See `libc_db`(3LIB). For information on `librtld_db`, refer to the *Linker and Libraries Guide*.

Refer to the individual reference manual pages that describe these routines for a functional specification that clients of `libthread_db` and `librtld_db` can use to implement this required interface. The <proc_service.h> header lists the C declarations of these routines.

Functions

`ps_pdmodel()`	Returns the data model of the target process.
`ps_pglobal_lookup()`	Looks up the symbol in the symbol table of the load object in the target process and returns its address.
`ps_pglobal_sym()`	Looks up the symbol in the symbol table of the load object in the target process and returns its symbol table entry.
`ps_pread()`	Copies *size* bytes from the target process to the controlling process.
`ps_pwrite()`	Copies *size* bytes from the controlling process to the target process.
`ps_pdread()`	Identical to `ps_pread()`.
`ps_pdwrite()`	Identical to `ps_pwrite()`.
`ps_ptread()`	Identical to `ps_pread()`.
`ps_ptwrite()`	Identical to `ps_pwrite()`.
`ps_pstop()`	Stops the target process.
`ps_pcontinue()`	Resumes target process.

`ps_lstop()`	Stops a single lightweight process (LWP) within the target process.
`ps_lcontinue()`	Resumes a single LWP within the target process.
`ps_lgetregs()`	Gets the general registers of the LWP.
`ps_lsetregs()`	Sets the general registers of the LWP.
`ps_lgetfpregs()`	Gets the LWP's floating point register set.
`ps_lsetfpregs()`	Sets the LWP's floating point register set.
`ps_pauxv()`	Returns a pointer to a read-only copy of the target process's auxiliary vector.
`ps_kill()`	Sends signal to target process.
`ps_lrolltoaddr()`	Rolls the LWP out of a critical section when the process is stopped.
`ps_plog()`	Logs a message.

SPARC

`ps_lgetxregsize()`	Returns the size of the architecture-dependent extra state registers.
`ps_lgetxregs()`	Gets the extra state registers of the LWP.
`ps_lsetxregs()`	Sets the extra state registers of the LWP.

x86

`ps_lgetLDT()`	Reads the local descriptor table of the LWP.

Attributes

See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also

`libc_db(3LIB)`, `librtld_db(3LIB)`, `ps_pread(3PROC)`, `rtld_db(3EXT)`, `attributes(5)`

Linker and Libraries Guide

Name ps_lgetregs, ps_lsetregs, ps_lgetfpregs, ps_lsetfpregs, ps_lgetxregsize, ps_lgetxregs, ps_lsetxregs – routines that access the target process register in libthread_db

Synopsis

```
#include <proc_service.h>

ps_err_e ps_lgetregs(struct ps_prochandle *ph, lwpid_t lid,
     prgregset_t gregset);

ps_err_e ps_lsetregs(struct ps_prochandle *ph, lwpid_t lid,
     static prgregset_t gregset);

ps_err_e ps_lgetfpregs(struct ps_prochandle *ph, lwpid_t lid,
     prfpregset_t *fpregs);

ps_err_e ps_lsetfpregs(struct ps_prochandle *ph, lwpid_t lid,
     static prfpregset_t *fpregs);

ps_err_e ps_lgetxregsize(struct ps_prochandle *ph, lwpid_t lid,
     int *xregsize);

ps_err_e ps_lgetxregs(struct ps_prochandle *ph, lwpid_t lid,
     caddr_t xregset);

ps_err_e ps_lsetxregs(struct ps_prochandle *ph, lwpid_t lid,
     caddr_t xregset);
```

Description `ps_lgetregs()`, `ps_lsetregs()`, `ps_lgetfpregs()`, `ps_lsetfpregs()`, `ps_lgetxregsize()`, `ps_lgetxregs()`, `ps_lsetxregs()` read and write register sets from lightweight processes (LWPs) within the target process identified by *ph*. `ps_lgetregs()` gets the general registers of the LWP identified by *lid*, and `ps_lsetregs()` sets them. `ps_lgetfpregs()` gets the LWP's floating point register set, while `ps_lsetfpregs()` sets it.

SPARC Only `ps_lgetxregsize()`,`ps_lgetxregs()`, and`ps_lsetxregs()` are SPARC-specific. They do not need to be defined by a controlling process on non-SPARC architecture. `ps_lgetxregsize()` returns in **xregsize* the size of the architecture-dependent extra state registers. `ps_lgetxregs()` gets the extra state registers, and `ps_lsetxregs()` sets them.

Return Values

`PS_OK` The call returned successfully.

`PS_NOFPREGS` Floating point registers are neither available for this architecture nor for this process.

`PS_NOXREGS` Extra state registers are not available on this architecture.

`PS_ERR` The function did not return successfully.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also libc_db(3LIB), proc_service(3PROC), attributes(5), threads(5)

Name ps_pglobal_lookup, ps_pglobal_sym – look up a symbol in the symbol table of the load object in the target process

Synopsis

```
#include <proc_service.h>

ps_err_e ps_pglobal_lookup(struct ps_prochandle *ph,
     const char *object_name, const char *sym_name,
     psaddr_t *sym_addr);

ps_err_e ps_pglobal_sym(struct ps_prochandle *ph,
     const char *object_name, const char *sym_name,
     ps_sym_t *sym);
```

Description `ps_pglobal_lookup()` looks up the symbol *sym_name* in the symbol table of the load object *object_name* in the target process identified by *ph*. It returns the symbol's value as an address in the target process in **sym_addr*.

`ps_pglobal_sym()` looks up the symbol *sym_name* in the symbol table of the load object *object_name* in the target process identified by *ph*. It returns the symbol table entry in **sym*. The value in the symbol table entry is the symbol's value as an address in the target process.

Return Values

`PS_OK` The call completed successfully.

`PS_NOSYM` The specified symbol was not found.

`PS_ERR` The function did not return successfully.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also `kill(2)`, `libc_db(3LIB)`, `proc_service(3PROC)`, `attributes(5)`, `threads(5)`

Name ps_pread, ps_pwrite, ps_pdread, ps_pdwrite, ps_ptread, ps_ptwrite – interfaces in libthread_db that target process memory access

Synopsis

```
#include <proc_service.h>

ps_err_e ps_pread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_pwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_pdread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_pdwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);

ps_err_e ps_ptread(struct ps_prochandle *ph, psaddr_t addr,
     void *buf, size_t size);

ps_err_e ps_ptwrite(struct ps_prochandle *ph, psaddr_t addr,
     const void *buf, size_t size);
```

Description These routines copy data between the target process's address space and the controlling process. `ps_pread()` copies *size* bytes from address *addr* in the target process into *buf* in the controlling process. `pr_pwrite()` is like `ps_pread()` except that the direction of the copy is reversed; data is copied from the controlling process to the target process.

`ps_pdread()` and `ps_ptread()` behave identically to `ps_pread()`. `ps_pdwrite()` and `ps_ptwrite()` behave identically to `ps_pwrite()`. These functions can be implemented as simple aliases for the corresponding primary functions. They are artifacts of history that must be maintained.

Return Values

`PS_OK` The call returned successfully. *size* bytes were copied.

`PS_BADADDR` Some part of the address range from *addr* through *addr*+*size*–1 is not part of the target process's address space.

`PS_ERR` The function did not return successfully.

Attributes See `attributes(5)` for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also `libc_db(3LIB)`, `librtld_db(3LIB)`, `proc_service(3PROC)`, `rtld_db(3EXT)`, `attributes(5)`, `threads(5)`

Name ps_pstop, ps_pcontinue, ps_lstop, ps_lcontinue, ps_lrolltoaddr, ps_kill – process and LWP control in libthread_db

Synopsis

```
#include <proc_service.h>

ps_err_e ps_pstop(struct ps_prochandle *ph);

ps_err_e ps_pcontinue(struct ps_prochandle *ph);

ps_err_e ps_lstop(struct ps_prochandle *ph, lwpid_t lwpid);

ps_err_e ps_lcontinue(struct ps_prochandle *ph,
     lwpid_t lwpid);

ps_err_e ps_lrolltoaddr(struct ps_prochandle *ph,
     lwpid_t lwpid, psaddr_t go_addr, psaddr_t stop_addr);

ps_err_e ps_kill(struct ps_prochandle *ph, int signum);
```

Description The `ps_pstop()` function stops the target process identified by *ph*, while the `ps_pcontinue()` function allows it to resume.

The `libthread_db()` function uses `ps_pstop()` to freeze the target process while it is under inspection. Within the scope of any single call from outside `libthread_db` to a `libthread_db` routine, `libthread_db` will call `ps_pstop()`, at most once. If it does, it will call `ps_pcontinue()` within the scope of the same routine.

The controlling process may already have stopped the target process when it calls `libthread_db`. In that case, it is not obligated to resume the target process when `libthread_db` calls `ps_pcontinue()`. In other words, `ps_pstop()` is mandatory, while `ps_pcontinue()` is advisory. After `ps_pstop()`, the target process must be stopped; after `ps_pcontinue()`, the target process may be running.

The `ps_lstop()` and `ps_lcontinue()` functions stop and resume a single lightweight process (LWP) within the target process *ph*.

The `ps_lrolltoaddr()` function is used to roll an LWP forward out of a critical section when the process is stopped. It is also used to run the `libthread_db` agent thread on behalf of `libthread`. The `ps_lrolltoaddr()` function is always called with the target process stopped, that is, there has been a preceding call to `ps_pstop()`. The specified LWP must be continued at the address *go_addr*, or at its current address if *go_addr* is `NULL`. It should then be stopped when its execution reaches *stop_addr*. This routine does not return until the LWP has stopped at *stop_addr*.

The `ps_kill()` function directs the signal *signum* to the target process for which the handle is *ph*. It has the same semantics as `kill(2)`.

Return Values

PS_OK — The call completed successfully. In the case of `ps_pstop()`, the target process is stopped.

PS_BADLID — For `ps_lstop()`, `ps_lcontinue()` and `ps_lrolltoaddr()`; there is no LWP with id *lwipd* in the target process.

PS_ERR — The function did not return successfully.

Attributes

See attributes(5) for description of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
MT Level	Safe

See Also

kill(2), libc_db(3LIB), proc_service(3PROC), attributes(5), threads(5)

Name sched_getparam – get scheduling parameters

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <sched.h>

int sched_getparam(pid_t pid, struct sched_param *param);
```

Description The `sched_getparam()` function returns the scheduling parameters of a process specified by *pid* in the `sched_param` structure pointed to by *param*.

If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters for the process whose process ID is equal to *pid* will be returned.

If *pid* is 0, the scheduling parameters for the calling process will be returned. The behavior of the `sched_getparam()` function is unspecified if the value of *pid* is negative.

Return Values Upon successful completion, the `sched_getparam()` function returns `0`. If the call to `sched_getparam()` is unsuccessful, the function returns −1 and sets `errno` to indicate the error.

Errors The `sched_getparam()` function will fail if:

`ENOSYS` The `sched_getparam()` function is not supported by the system.

`EPERM` The requesting process does not have permission to obtain the scheduling parameters of the specified process.

`ESRCH` No process can be found corresponding to that specified by *pid*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `librt(3LIB)`, `sched.h(3HEAD)`, `sched_getscheduler(3RT)`, `sched_setparam(3RT)`, `sched_setscheduler(3RT)`, `attributes(5)`

Notes Solaris 2.6 was the first release to support `libposix4/librt`. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name sched_get_priority_max, sched_get_priority_min – get scheduling parameter limits

Synopsis
```
cc [ flag... ] file... -lrt [ library... ]
#include <sched.h>

int sched_get_priority_max(int policy);

int sched_get_priority_min(int policy);
```

Description The `sched_get_priority_max()` and `sched_get_priority_min()` functions return the appropriate maximum or minimum, respectfully, for the scheduling policy specified by *policy*.

The value of *policy* is one of the scheduling policy values defined in `<sched.h>`.

Return Values If successful, the `sched_get_priority_max()` and `sched_get_priority_min()` functions return the appropriate maximum or minimum values, respectively. If unsuccessful, they return −1 and set `errno` to indicate the error.

Errors The `sched_get_priority_max()` and `sched_get_priority_min()` functions will fail if:

`EINVAL` The value of the *policy* parameter does not represent a defined scheduling policy.

`ENOSYS` The `sched_get_priority_max()`, `sched_get_priority_min()` and `sched_rr_get_interval`(3RT) functions are not supported by the system.

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `librt`(3LIB), `sched.h`(3HEAD), `sched_getparam`(3RT), `sched_setparam`(3RT), `sched_getscheduler`(3RT), `sched_rr_get_interval`(3RT), `sched_setscheduler`(3RT), `time.h`(3HEAD), `attributes`(5)

Notes Solaris 2.6 was the first release to support `libposix4/librt`. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name sched_getscheduler – get scheduling policy

Synopsis
```
cc [ flag... ] file... -lrt [ library... ]
#include <sched.h>

int sched_getscheduler(pid_t pid);
```

Description The `sched_getscheduler()` function returns the scheduling policy of the process specified by *pid*. If the value of *pid* is negative, the behavior of the `sched_getscheduler()` function is unspecified.

The values that can be returned by `sched_getscheduler()` are defined in the header `<sched.h>` and described on the `sched_setscheduler(3RT)` manual page.

If a process specified by *pid* exists and if the calling process has permission, the scheduling policy will be returned for the process whose process ID is equal to *pid*.

If *pid* is 0, the scheduling policy will be returned for the calling process.

Return Values Upon successful completion, the `sched_getscheduler()` function returns the scheduling policy of the specified process. If unsuccessful, the function returns −1 and sets `errno` to indicate the error.

Errors The `sched_getscheduler()` function will fail if:

`ENOSYS` The `sched_getscheduler()` function is not supported by the system.

`EPERM` The requesting process does not have permission to determine the scheduling policy of the specified process.

`ESRCH` No process can be found corresponding to that specified by *pid*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `libr t(3LIB)`, `sched.h(3HEAD)`, `sched_getparam(3RT)`, `sched_setparam(3RT)`, `sched_setscheduler(3RT)`, `attributes(5)`

Notes Solaris 2.6 was the first release to support `libposix4`/`librt`. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name sched_rr_get_interval – get execution time limits

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <sched.h>

int sched_rr_get_interval(pid_t pid,
     struct timespec *interval);
```

Description The `sched_rr_get_interval()` function updates the `timespec` structure referenced by the *interval* argument to contain the current execution time limit (that is, time quantum) for the process specified by *pid*. If *pid* is 0, the current execution time limit for the calling process will be returned.

Return Values If successful, the `sched_rr_get_interval()` function returns `0`. Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `sched_rr_get_interval()` function will fail if:

`ENOSYS` The `sched_get_priority_max(3RT)`, `sched_get_priority_min(3RT)`, and `sched_rr_get_interval()` functions are not supported by the system.

`ESRCH` No process can be found corresponding to that specified by *pid*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `librt(3LIB)`, `sched.h(3HEAD)`, `sched_getparam(3RT)`, `sched_setparam(3RT)`, `sched_get_priority_max(3RT)`, `sched_getscheduler(3RT)`, `sched_setscheduler(3RT)`, `attributes(5)`

Notes Solaris 2.6 was the first release to support `libposix4/librt`. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name sched_setparam – set scheduling parameters

Synopsis
```
cc [ flag... ] file... -lrt [ library... ]
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
```

Description The `sched_setparam()` function sets the scheduling parameters of the process specified by *pid* to the values specified by the `sched_param` structure pointed to by *param*. The value of the *sched_priority* member in the `sched_param` structure is any integer within the inclusive priority range for the current scheduling policy of the process specified by *pid*. Higher numerical values for the priority represent higher priorities. If the value of *pid* is negative, the behavior of the `sched_setparam()` function is unspecified.

If a process specified by *pid* exists and if the calling process has permission, the scheduling parameters will be set for the process whose process ID is equal to *pid*. The real or effective user ID of the calling process must match the real or saved (from `exec(2)`) user ID of the target process unless the effective user ID of the calling process is `0`. See `Intro(2)`.

If *pid* is zero, the scheduling parameters will be set for the calling process.

The target process, whether it is running or not running, resumes execution after all other runnable processes of equal or greater priority have been scheduled to run.

If the priority of the process specified by the *pid* argument is set higher than that of the lowest priority running process and if the specified process is ready to run, the process specified by the *pid* argument preempts a lowest priority running process. Similarly, if the process calling `sched_setparam()` sets its own priority lower than that of one or more other non-empty process lists, then the process that is the head of the highest priority list also preempts the calling process. Thus, in either case, the originating process might not receive notification of the completion of the requested priority change until the higher priority process has executed.

If the current scheduling policy for the process specified by *pid* is not `SCHED_FIFO` or `SCHED_RR`, including `SCHED_OTHER`, the result is equal to `priocntl(P_PID`, *pid*, `PC_SETPARMS`, &*pcparam*), where *pcparam* is an image of **param*.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:

- For threads with system scheduling contention scope, these functions have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads' scheduling parameters will not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel scheduled entities, then the underlying kernel scheduled entities for the system contention scope threads will not be affected by these functions.

The underlying kernel scheduled entities for the process contention scope threads will have their scheduling parameters changed to the value specified in *param*. Kernel scheduled entities for use by process contention scope threads that are created after this call completes inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads are allowed to continue to execute while this function call is in the process of changing the scheduling policy for the underlying kernel scheduled entities used by the process contention scope threads.

Return Values If successful, the `sched_setparam()` function returns `0`.

If the call to `sched_setparam()` is unsuccessful, the priority remains unchanged, and the function returns `−1` and sets `errno` to indicate the error.

Errors The `sched_setparam()` function will fail if:

`EINVAL` One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified *pid*.

`ENOSYS` The `sched_setparam()` function is not supported by the system.

`EPERM` The requesting process does not have permission to set the scheduling parameters for the specified process, or does not have the appropriate privilege to invoke `sched_setparam()`.

`ESRCH` No process can be found corresponding to that specified by *pid*.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `Intro(2)`, `exec(2)`, `librt(3LIB)`, `sched.h(3HEAD)`, `sched_getparam(3RT)`, `sched_getscheduler(3RT)`, `sched_setscheduler(3RT)`, `attributes(5)`

Notes Solaris 2.6 was the first release to support `libposix4/librt`. Prior to this release, this function always returned `−1` and set `errno` to `ENOSYS`.

Name sched_setscheduler – set scheduling policy and scheduling parameters

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
     const struct sched_param *param);
```

Description The `sched_setscheduler()` function sets the scheduling policy and scheduling parameters of the process specified by *pid* to *policy* and the parameters specified in the `sched_param` structure pointed to by *param*, respectively. The value of the `sched_priority` member in the `sched_param` structure is any integer within the inclusive priority range for the scheduling policy specified by *policy*. The `sched_setscheduler()` function ignores the other members of the `sched_param` structure. If the value of *pid* is negative, the behavior of the `sched_setscheduler()` function is unspecified.

The possible values for the *policy* parameter are defined in the header `<sched.h>` (see `sched.h`(3HEAD)):

If a process specified by *pid* exists and if the calling process has permission, the scheduling policy and scheduling parameters are set for the process whose process ID is equal to *pid.* The real or effective user ID of the calling process must match the real or saved (from `exec`(2)) user ID of the target process unless the effective user ID of the calling process is 0. See `Intro`(2).

If *pid* is 0, the scheduling policy and scheduling parameters are set for the calling process.

To change the *policy* of any process to either of the real time policies `SCHED_FIFO` or `SCHED_RR`, the calling process must either have the `SCHED_FIFO` or `SCHED_RR` policy or have an effective user ID of `0`.

The `sched_setscheduler()` function is considered successful if it succeeds in setting the scheduling policy and scheduling parameters of the process specified by *pid* to the values specified by *policy* and the structure pointed to by *param*, respectively.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:

- For threads with system scheduling contention scope, these functions have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads' scheduling policy and associated parameters will not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

The underlying kernel scheduled entities for the process contention scope threads will have their scheduling policy and associated scheduling parameters changed to the values specified in *policy* and *param*, respectively. Kernel scheduled entities for use by process contention scope threads that are created after this call completes inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads are allowed to continue to execute while this function call is in the process of changing the scheduling policy and associated scheduling parameters for the underlying kernel scheduled entities used by the process contention scope threads.

Return Values Upon successful completion, the function returns the former scheduling policy of the specified process. If the `sched_setscheduler()` function fails to complete successfully, the policy and scheduling paramenters remain unchanged, and the function returns −1 and sets `errno` to indicate the error.

Errors The `sched_setscheduler()` function will fail if:

`EINVAL` The value of *policy* is invalid, or one or more of the parameters contained in `param` is outside the valid range for the specified scheduling policy.

`ENOSYS` The `sched_setscheduler()` function is not supported by the system.

`EPERM` The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.

`ESRCH` No process can be found corresponding to that specified by *pid.*

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `priocntl(1)`, `Intro(2)`, `exec(2)`, `priocntl(2)`, `librt(3LIB)`, `sched.h(3HEAD)`, `sched_get_priority_max(3RT)`, `sched_getparam(3RT)`, `sched_getscheduler(3RT)`, `sched_setparam(3RT)`, `attributes(5)`

Notes Solaris 2.6 was the first release to support `libposix4/librt`. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name sched_yield – yield processor

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <sched.h>

int sched_yield(void);
```

Description The `sched_yield()` function forces the running thread to relinquish the processor until the process again becomes the head of its process list. It takes no arguments.

Return Values If successful, `sched_yield()` returns `0`, otherwise, it returns `−1`, and sets `errno` to indicate the error condition.

Errors No errors are defined.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `librt(3LIB)`, `sched.h(3HEAD)`, `attributes(5)`

Name sem_close – close a named semaphore

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Description The `sem_close()` function is used to indicate that the calling process is finished using the named semaphore indicated by *sem*. The effects of calling `sem_close()` for an unnamed semaphore (one created by `sem_init`(3RT)) are undefined. The `sem_close()` function deallocates (that is, make available for reuse by a subsequent `sem_open`(3RT) by this process) any system resources allocated by the system for use by this process for this semaphore. The effect of subsequent use of the semaphore indicated by *sem* by this process is undefined. If the semaphore has not been removed with a successful call to `sem_unlink`(3RT), then `sem_close()` has no effect on the state of the semaphore. If the `sem_unlink`(3RT) function has been successfully invoked for *name* after the most recent call to `sem_open`(3RT) with `O_CREAT` for this semaphore, then when all processes that have opened the semaphore close it, the semaphore is no longer be accessible.

Return Values If successful, `sem_close()` returns `0`, otherwise it returns −1 and sets `errno` to indicate the error.

Errors The `sem_close()` function will fail if:

`EINVAL` The *sem* argument is not a valid semaphore descriptor.

`ENOSYS` The `sem_close()` function is not supported by the system.

Usage The `sem_close()` function should not be called for an unnamed semaphore initialized by `sem_init`(3RT).

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `sem_init`(3RT), `sem_open`(3RT), `sem_unlink`(3RT), `attributes`(5), `standards`(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name sem_destroy – destroy an unnamed semaphore

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Description The `sem_destroy()` function is used to destroy the unnamed semaphore indicated by *sem*. Only a semaphore that was created using `sem_init(3RT)` may be destroyed using `sem_destroy()`; the effect of calling `sem_destroy()` with a named semaphore is undefined. The effect of subsequent use of the semaphore *sem* is undefined until *sem* is re-initialized by another call to `sem_init(3RT)`.

It is safe to destroy an initialised semaphore upon which no threads are currently blocked. The effect of destroying a semaphore upon which other threads are currently blocked is undefined.

Return Values If successful, `sem_destroy()` returns `0`, otherwise it returns −1 and sets `errno` to indicate the error.

Errors The `sem_destroy()` function will fail if:

`EINVAL` The *sem* argument is not a valid semaphore.

The `sem_destroy()` function may fail if:

`EBUSY` There are currently processes (or LWPs or threads) blocked on the semaphore.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `sem_init(3RT)`, `sem_open(3RT)`, `attributes(5)`, `standards(5)`

Name sem_getvalue – get the value of a semaphore

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>

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

Description The `sem_getvalue()` function updates the location referenced by the *sval* argument to have the value of the semaphore referenced by *sem* without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but it need not be the actual value of the semaphore when it is returned to the calling process.

If *sem* is locked, then the value returned by `sem_getvalue()` is either zero or a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call.

The value set in *sval* may be 0 or positive. If *sval* is 0, there may be other processes (or LWPs or threads) waiting for the semaphore; if *sval* is positive, no process is waiting.

Return Values Upon successful completion, `sem_getvalue()` returns `0`. Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `sem_getvalue()` function will fail if:

`EINVAL` The *sem* argument does not refer to a valid semaphore.

`ENOSYS` The `sem_getvalue()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `semctl(2)`, `semget(2)`, `semop(2)`, `sem_post(3RT)`, `sem_wait(3RT)`, `attributes(5)`, `standards(5)`

Name sem_init – initialize an unnamed semaphore

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>

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

Description The `sem_init()` function is used to initialize the unnamed semaphore referred to by *sem*. The value of the initialized semaphore is *value*. Following a successful call to `sem_init()`, the semaphore may be used in subsequent calls to `sem_wait(3RT)`, `sem_trywait(3RT)`, `sem_post(3RT)`, and `sem_destroy(3RT)`. This semaphore remains usable until the semaphore is destroyed.

If the *pshared* argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore *sem* can use *sem* for performing `sem_wait(3RT)`, `sem_trywait(3RT)`, `sem_post(3RT)`, and `sem_destroy(3RT)` operations.

Only *sem* itself may be used for performing synchronization. The result of referring to copies of *sem* in calls to `sem_wait(3RT)`, `sem_trywait(3RT)`, `sem_post(3RT)`, and `sem_destroy(3RT)`, is undefined.

If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use *sem* for performing `sem_wait(3RT)`, `sem_trywait(3RT)`, `sem_post(3RT)`, and `sem_destroy(3RT)` operations. The use of the semaphore by threads other than those created in the same process is undefined.

Attempting to initialize an already initialized semaphore results in undefined behavior.

Return Values Upon successful completion, the function initializes the semaphore in *sem*. Otherwise, it returns −1 and sets `errno` to indicate the error.

Errors The `sem_init()` function will fail if:

`EINVAL` The *value* argument exceeds `SEM_VALUE_MAX`.

`ENOSPC` A resource required to initialize the semaphore has been exhausted, or the resources have reached the limit on semaphores (`SEM_NSEMS_MAX`).

`ENOSYS` The `sem_init()` function is not supported by the system.

`EPERM` The process lacks the appropriate privileges to initialize the semaphore.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also sem_destroy(3RT), sem_post(3RT), sem_wait(3RT), attributes(5), standards(5)

Name sem_open – initialize/open a named semaphore

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag,
     /* unsigned long mode, unsigned int value */ ...);
```

Description The `sem_open()` function establishes a connection between a named semaphore and a process (or LWP or thread). Following a call to `sem_open()` with semaphore name *name*, the process may reference the semaphore associated with *name* using the address returned from the call. This semaphore may be used in subsequent calls to `sem_wait`(3RT), `sem_trywait`(3RT), `sem_post`(3RT), and `sem_close`(3RT). The semaphore remains usable by this process until the semaphore is closed by a successful call to `sem_close`(3RT), `_Exit`(2), or one of the `exec` functions.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to `sem_open()`. The following flag bits may be set in *oflag*:

`O_CREAT` This flag is used to create a semaphore if it does not already exist. If `O_CREAT` is set and the semaphore already exists, then `O_CREAT` has no effect, except as noted under `O_EXCL.` Otherwise, `sem_open()` creates a named semaphore. The `O_CREAT` flag requires a third and a fourth argument: *mode*, which is of type `mode_t`, and *value*, which is of type `unsigned int`. The semaphore is created with an initial value of *value*. Valid initial values for semaphores are less than or equal to `SEM_VALUE_MAX.`

The user ID of the semaphore is set to the effective user ID of the process; the group ID of the semaphore is set to a system default group ID or to the effective group ID of the process. The permission bits of the semaphore are set to the value of the *mode* argument except those set in the file mode creation mask of the process (see `umask`(2)). When bits in *mode* other than the file permission bits are specified, the effect is unspecified.

After the semaphore named *name* has been created by `sem_open()` with the `O_CREAT` flag, other processes can connect to the semaphore by calling `sem_open()` with the same value of *name*.

`O_EXCL` If `O_EXCL` and `O_CREAT` are set, `sem_open()` fails if the semaphore *name* exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing `sem_open()` with `O_EXCL` and `O_CREAT` set. If `O_EXCL` is set and `O_CREAT` is not set, the effect is undefined.

If flags other than `O_CREAT` and `O_EXCL` are specified in the *oflag* parameter, the effect is unspecified.

The *name* argument points to a string naming a semaphore object. It is unspecified whether the name appears in the file system and is visible to functions that take pathnames as arguments. The *name* argument conforms to the construction rules for a pathname. The first character of *name* must be a slash (/) character and the remaining characters of *name* cannot include any slash characters. For maximum portability, *name* should include no more than 14 characters, but this limit is not enforced.

If a process makes multiple successful calls to `sem_open()` with the same value for *name*, the same semaphore address is returned for each such successful call, provided that there have been no calls to `sem_unlink`(3RT) for this semaphore.

References to copies of the semaphore produce undefined results.

Return Values Upon successful completion, the function returns the address of the semaphore. Otherwise, it will return a value of `SEM_FAILED` and set `errno` to indicate the error. The symbol `SEM_FAILED` is defined in the header `<semaphore.h>`. No successful return from `sem_open()` will return the value `SEM_FAILED.`

Errors If any of the following conditions occur, the `sem_open()` function will return `SEM_FAILED` and set `errno` to the corresponding value:

`EACCES`	The named semaphore exists and the `O_RDWR` permissions are denied, or the named semaphore does not exist and permission to create the named semaphore is denied.
`EEXIST`	`O_CREAT` and `O_EXCL` are set and the named semaphore already exists.
`EINTR`	The `sem_open()` function was interrupted by a signal.
`EINVAL`	The `sem_open()` operation is not supported for the given name, or `O_CREAT` was set in *oflag* and *value* is greater than `SEM_VALUE_MAX`.
`EMFILE`	The number of open semaphore descriptors in this process exceeds `SEM_NSEMS_MAX`, or the number of open file descriptors in this process exceeds `OPEN_MAX`.
`ENAMETOOLONG`	The length of *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.
`ENFILE`	Too many semaphores are currently open in the system.
`ENOENT`	`O_CREAT` is not set and the named semaphore does not exist.
`ENOSPC`	There is insufficient space for the creation of the new named semaphore.
`ENOSYS`	The `sem_open()` function is not supported by the system.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also exec(2), exit(2), umask(2), sem_close(3RT), sem_post(3RT), sem_unlink(3RT), sem_wait(3RT), sysconf(3C), attributes(5), standards(5)

Name sem_post – increment the count of a semaphore

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Description The `sem_post()` function unlocks the semaphore referenced by *sem* by performing a semaphore unlock operation on that semaphore.

If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this operation is 0, then one of the threads blocked waiting for the semaphore will be allowed to return successfully from its call to `sem_wait`(3RT). If the symbol `_POSIX_PRIORITY_SCHEDULING` is defined, the thread to be unblocked will be chosen in a manner appropriate to the scheduling policies and parameters in effect for the blocked threads. In the case of the schedulers `SCHED_FIFO` and `SCHED_RR`, the highest priority waiting thread will be unblocked, and if there is more than one highest priority thread blocked waiting for the semaphore, then the highest priority thread that has been waiting the longest will be unblocked. If the symbol `_POSIX_PRIORITY_SCHEDULING` is not defined, the choice of a thread to unblock is unspecified.

Return Values If successful, `sem_post()` returns `0`; otherwise it returns `−1` and sets `errno` to indicate the error.

Errors The `sem_post()` function will fail if:

`EINVAL` The *sem* argument does not refer to a valid semaphore.

`ENOSYS` The `sem_post()` function is not supported by the system.

`EOVERFLOW` The semaphore value exceeds `SEM_VALUE_MAX`.

Usage The `sem_post()` function is reentrant with respect to signals and may be invoked from a signal-catching function. The semaphore functionality described on this manual page is for the POSIX (see `standards`(5)) threads implementation. For the documentation of the Solaris threads interface, see `semaphore`(3C)).

Examples See `sem_wait`(3RT).

Attributes See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also sched_setscheduler(3RT), sem_wait(3RT), semaphore(3C), attributes(5), standards(5)

Name sem_timedwait, sem_reltimedwait_np – lock a semaphore

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
     const struct timespec *restrict abs_timeout);

int sem_reltimedwait_np(sem_t *restrict sem,
     const struct timespec *restrict rel_timeout);
```

Description The `sem_timedwait()` function locks the semaphore referenced by *sem* as in the `sem_wait(3RT)` function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a `sem_post(3RT)` function, this wait is terminated when the specified timeout expires.

The `sem_reltimedwait_np()` function is identical to the `sem_timedwait()` function, except that the timeout is specified as a relative time interval.

For `sem_timedwait()`, the timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` clock (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

For `sem_reltimedwait_np()`, the timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` data type is defined as a structure in the `<time.h>` header.

Under no circumstance does the function fail with a timeout if the semaphore can be locked immediately. The validity of the *abs_timeout* need not be checked if the semaphore can be locked immediately.

Return Values The `sem_timedwait()` and `sem_reltimedwait_np()` functions return 0 if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore is be unchanged and the function returns -1 and sets `errno` to indicate the error.

Errors The `sem_timedwait()` and `sem_reltimedwait_np()` functions will fail if:

`EINVAL` The *sem* argument does not refer to a valid semaphore.

`EINVAL` The process or thread would have blocked, and the timeout parameter specified a nanoseconds field value less than zero or greater than or equal to 1,000 million.

`ETIMEDOUT` The semaphore could not be locked before the specified timeout expired.

The `sem_timedwait()` and `sem_reltimedwait_np()` functions may fail if:

`EDEADLK` A deadlock condition was detected.

`EINTR` A signal interrupted this function.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	MT-Safe

The `sem_timedwait()` is function Standard. The `sem_reltimedwait_np()` function is Stable.

See Also `semctl(2)`, `semget(2)`, `semop(2)`, `time(2)`, `sem_post(3RT)`, `sem_trywait(3RT)sem_wait(3RT)`, `attributes(5)`, `standards(5)`

Name sem_unlink – remove a named semaphore

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>

int sem_unlink(const char *name);
```

Description The `sem_unlink()` function removes the semaphore named by the string *name*. If the semaphore named by *name* is currently referenced by other processes, then `sem_unlink()` has no effect on the state of the semaphore. If one or more processes have the semaphore open when `sem_unlink()` is called, destruction of the semaphore is postponed until all references to the semaphore have been destroyed by calls to `sem_close(3RT)`, `_Exit(2)`, or one of the `exec` functions (see `exec(2)`) . Calls to `sem_open(3RT)` to re-create or re-connect to the semaphore refer to a new semaphore after `sem_unlink()` is called. The `sem_unlink()` call does not block until all references have been destroyed; it returns immediately.

Return Values Upon successful completion, `sem_unlink()` returns `0`. Otherwise, the semaphore is not changed and the function returns a value of −1 and sets `errno` to indicate the error.

Errors The `sem_unlink()` function will fail if:

`EACCES` Permission is denied to unlink the named semaphore.

`ENAMETOOLONG` The length of *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

`ENOENT` The named semaphore does not exist.

`ENOSYS` The `sem_unlink()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `exec(2)`, `exit(2)`, `sem_close(3RT)`, `sem_open(3RT)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set `errno` to `ENOSYS`.

Name sem_wait, sem_trywait – acquire or wait for a semaphore

Synopsis
```
cc [ flag... ] file... -lrt [ library... ]
#include <semaphore.h>

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);
```

Description The `sem_wait()` function locks the semaphore referenced by *sem* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread will not return from the call to `sem_wait()` until it either locks the semaphore or the call is interrupted by a signal. The `sem_trywait()` function locks the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.

Upon successful return, the state of the semaphore is locked and remains locked until the `sem_post(3RT)` function is executed and returns successfully.

The `sem_wait()` function is interruptible by the delivery of a signal.

Return Values The `sem_wait()` and `sem_trywait()` functions return `0` if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore is unchanged, and the function returns `−1` and sets `errno` to indicate the error.

Errors The `sem_wait()` and `sem_trywait()` functions will fail if:

`EINVAL` The *sem* function does not refer to a valid semaphore.

`ENOSYS` The `sem_wait()` and `sem_trywait()` functions are not supported by the system.

The `sem_trywait()` function will fail if:

`EAGAIN` The semaphore was already locked, so it cannot be immediately locked by the `sem_trywait()` operation.

The `sem_wait()` and `sem_trywait()` functions may fail if:

`EDEADLK` A deadlock condition was detected; that is, two separate processes are waiting for an available resource to be released via a semaphore "held" by the other process.

`EINTR` A signal interrupted this function.

Usage Realtime applications may encounter priority inversion when using semaphores. The problem occurs when a high priority thread “locks” (that is, waits on) a semaphore that is about to be “unlocked” (that is, posted) by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by semaphores execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

Examples

EXAMPLE 1 The customer waiting-line in a bank may be analogous to the synchronization scheme of a semaphore utilizing `sem_wait()` and `sem_trywait()`:

```
#include <errno.h>
#define TELLERS 10
sem_t  bank_line;      /* semaphore */
int banking_hours(), deposit_withdrawal;
void *customer(), do_business(), skip_banking_today();
thread_t tid;
...

sem_init(&bank_line,TRUE,TELLERS);  /* 10 tellers
                                       available */
while(banking_hours())
        thr_create(NULL, NULL, customer,
            (void *)deposit_withdrawal, THREAD_NEW_LWP, &tid);
...

void *
customer(deposit_withdrawal)
void *deposit_withdrawal;
{
    int this_customer, in_a_hurry = 50;
    this_customer = rand() % 100;
    if (this_customer == in_a_hurry)  {
        if (sem_trywait(&bank_line) != 0)
        if (errno == EAGAIN) {    /* no teller available */
            skip_banking_today(this_customer);
            return;
        }     /*else go immediately to available teller
               & decrement bank_line*/
    }
    else
        sem_wait(&bank_line); /* wait for next teller,
             then proceed, and decrement bank_line */
    do_business((int *)deposit_withdrawal);
    sem_getvalue(&bank_line,&num_tellers);
    sem_post(&bank_line); /* increment bank_line;
        this_customer's teller is now available */
}
```

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also sem_post(3RT), attributes(5), standards(5)

Name shm_open – open a shared memory object

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

Description The `shm_open()` function establishes a connection between a shared memory object and a file descriptor. It creates an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that shared memory object. The *name* argument points to a string naming a shared memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument conforms to the construction rules for a pathname. The first character of *name* must be a slash (/) character and the remaining characters of *name* cannot include any slash characters. For maximum portability, *name* should include no more than 14 characters, but this limit is not enforced.

If successful, `shm_open()` returns a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other processes. It is unspecified whether the file offset is set. The `FD_CLOEXEC` file descriptor flag associated with the new file descriptor is set.

The file status flags and file access modes of the open file description are according to the value of *oflag*. The *oflag* argument is the bitwise inclusive OR of the following flags defined in the header `<fcntl.h>`. Applications specify exactly one of the first two values (access modes) below in the value of *oflag*:

`O_RDONLY` Open for read access only.

`O_RDWR` Open for read or write access.

Any combination of the remaining flags may be specified in the value of *oflag*:

`O_CREAT` If the shared memory object exists, this flag has no effect, except as noted under `O_EXCL` below. Otherwise the shared memory object is created; the user ID of the shared memory object will be set to the effective user ID of the process; the group ID of the shared memory object will be set to a system default group ID or to the effective group ID of the process. The permission bits of the shared memory object will be set to the value of the *mode* argument except those set in the file mode creation mask of the process. When bits in *mode* other than the file permission bits are set, the effect is unspecified. The *mode* argument does not affect whether the shared memory object is opened for reading, for writing, or for both. The shared memory object has a size of zero.

`O_EXCL` If `O_EXCL` and `O_CREAT` are set, `shm_open()` fails if the shared memory object exists. The check for the existence of the shared memory object and the creation of the object if it does not exist is atomic with respect to other processes executing `shm_open()` naming the same shared memory object with `O_EXCL` and `O_CREAT` set. If `O_EXCL` is set and `O_CREAT` is not set, the result is undefined.

`O_TRUNC` If the shared memory object exists, and it is successfully opened `O_RDWR`, the object will be truncated to zero length and the mode and owner will be unchanged by this function call. The result of using `O_TRUNC` with `O_RDONLY` is undefined.

When a shared memory object is created, the state of the shared memory object, including all data associated with the shared memory object, persists until the shared memory object is unlinked and all other references are gone. It is unspecified whether the name and shared memory object state remain valid after a system reboot.

Return Values Upon successful completion, the `shm_open()` function returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it returns −1 and sets `errno` to indicate the error condition.

Errors The `shm_open()` function will fail if:

`EACCES` The shared memory object exists and the permissions specified by *oflag* are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or `O_TRUNC` is specified and write permission is denied.

`EEXIST` `O_CREAT` and `O_EXCL` are set and the named shared memory object already exists.

`EINTR` The `shm_open()` operation was interrupted by a signal.

`EINVAL` The `shm_open()` operation is not supported for the given name.

`EMFILE` Too many file descriptors are currently in use by this process.

`ENAMETOOLONG` The length of the *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

`ENFILE` Too many shared memory objects are currently open in the system.

`ENOENT` `O_CREAT` is not set and the named shared memory object does not exist.

`ENOSPC` There is insufficient space for the creation of the new shared memory object.

`ENOSYS` The `shm_open()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE

Interface Stability	Standard
MT-Level	MT-Safe

See Also close(2), dup(2), exec(2), fcntl(2), mmap(2), umask(2), shm_unlink(3RT), sysconf(3C), fcntl.h(3HEAD), attributes(5), standards(5)

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −1 and set errno to ENOSYS.

Name shm_unlink – remove a shared memory object

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <sys/mman.h>

int shm_unlink(const char *name);
```

Description The `shm_unlink()` function removes the name of the shared memory object named by the string pointed to by *name*. If one or more references to the shared memory object exists when the object is unlinked, the name is removed before `shm_unlink()` returns, but the removal of the memory object contents will be postponed until all open and mapped references to the shared memory object have been removed.

Return Values Upon successful completion, `shm_unlink()` returns `0`. Otherwise it returns −`1` and sets `errno` to indicate the error condition, and the named shared memory object is not affected by this function call.

Errors The `shm_unlink()` function will fail if:

`EACCES` Permission is denied to unlink the named shared memory object.

`ENAMETOOLONG` The length of the *name* string exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

`ENOENT` The named shared memory object does not exist.

`ENOSYS` The `shm_unlink()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe

See Also `close(2)`, `mmap(2)`, `mlock(3C)`, `shm_open(3RT)`, `attributes(5)`, `standards(5)`

Notes Solaris 2.6 was the first release to support the Asynchronous Input and Output option. Prior to this release, this function always returned −`1` and set `errno` to `ENOSYS`.

Name sigqueue – queue a signal to a process

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <sys/types.h>
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);
```

Description The `sigqueue()` function causes the signal specified by *signo* to be sent with the value specified by *value* to the process specified by *pid*. If *signo* is 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*.

The conditions required for a process to have permission to queue a signal to another process are the same as for the `kill(2)` function.

The `sigqueue()` function returns immediately. If `SA_SIGINFO` is set for *signo* and if the resources were available to queue the signal, the signal is queued and sent to the receiving process. If `SA_SIGINFO` is not set for *signo*, then *signo* is sent at least once to the receiving process; it is unspecified whether *value* will be sent to the receiving process as a result of this call.

If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked for the calling thread and if no other thread has *signo* unblocked or is waiting in a `sigwait(2)` function for *signo*, either *signo* or at least the pending, unblocked signal will be delivered to the calling thread before the `sigqueue()` function returns. Should any of multiple pending signals in the range `SIGRTMIN` to `SIGRTMAX` be selected for delivery, it will be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

Return Values Upon successful completion, the specified signal will have been queued, and the `sigqueue()` function returns `0`. Otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `sigqueue()` function will fail if:

`EAGAIN` No resources are available to queue the signal. The process has already queued `SIGQUEUE_MAX` signals that are still pending at the receiver(s), or a system wide resource limit has been exceeded.

`EINVAL` The value of *signo* is an invalid or unsupported signal number.

`ENOSYS` The `sigqueue()` function is not supported by the system.

`EPERM` The process does not have the appropriate privilege to send the signal to the receiving process.

`ESRCH` The process *pid* does not exist.

Attributes See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also kill(2), siginfo.h(3HEAD), signal.h(3HEAD), sigwaitinfo(3RT), attributes(5), standards(5)

Name sigwaitinfo, sigtimedwait – wait for queued signals

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <signal.h>

int sigwaitinfo(const sigset_t *restrict set,
     siginfo_t *restrict info);

int sigtimedwait(const sigset_t *restrict set,
     siginfo_t *restrict info,
     const struct timespec *restrict timeout);
```

Description The `sigwaitinfo()` function selects the pending signal from the set specified by `set`. Should any of multiple pending signals in the range `SIGRTMIN` to `SIGRTMAX` be selected, it will be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified. If no signal in `set` is pending at the time of the call, the calling thread is suspended until one or more signals in `set` become pending or until it is interrupted by an unblocked, caught signal.

The `sigwaitinfo()` function behaves the same as the `sigwait(2)` function if the *info* argument is `NULL`. If the *info* argument is non-`NULL`, the `sigwaitinfo()` function behaves the same as `sigwait(2)`, except that the selected signal number is stored in the *si_signo* member, and the cause of the signal is stored in the *si_code* member. If any value is queued to the selected signal, the first such queued value is dequeued and, if the *info* argument is non-`NULL`, the value is stored in the *si_value* member of *info*. The system resource used to queue the signal will be released and made available to queue other signals. If no value is queued, the content of the *si_value* member is undefined. If no further signals are queued for the selected signal, the pending indication for that signal will be reset. If the value of the `si_code` member is `SI_NOINFO`, only the `si_signo` member of `siginfo_t` is meaningful, and the value of all other members is unspecified.

The `sigtimedwait()` function behaves the same as `sigwaitinfo()` except that if none of the signals specified by `set` are pending, `sigtimedwait()` waits for the time interval specified in the `timespec` structure referenced by *timeout*. If the `timespec` structure pointed to by *timeout* is zero-valued and if none of the signals specified by `set` are pending, then `sigtimedwait()` returns immediately with an error. If *timeout* is the `NULL` pointer, the behavior is unspecified.

If, while `sigwaitinfo()` or `sigtimedwait()` is waiting, a signal occurs which is eligible for delivery (that is, not blocked by the process signal mask), that signal is handled asynchronously and the wait is interrupted.

Return Values Upon successful completion (that is, one of the signals specified by *set* is pending or is generated) `sigwaitinfo()` and `sigtimedwait()` will return the selected signal number. Otherwise, the function returns −`1` and sets `errno` to indicate the error.

Errors The `sigwaitinfo()` and `sigtimedwait()` functions will fail if:

`EINTR` The wait was interrupted by an unblocked, caught signal.

`ENOSYS` The `sigwaitinfo()` and `sigtimedwait()` functions are not supported.

The `sigtimedwait()` function will fail if:

`EAGAIN` No signal specified by `set` was generated within the specified timeout period.

The `sigwaitinfo()` and `sigtimedwait()` functions may fail if:

`EFAULT` The *set*, *info*, or *timeout* argument points to an invalid address.

The `sigtimedwait()` function may fail if:

`EINVAL` The *timeout* argument specified a `tv_nsec` value less than zero or greater than or equal to 1000 million. The system only checks for this error if no signal is pending in *set* and it is necessary to wait.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Safe
Standard	See `standards(5)`.

See Also `time(2)`, `sigqueue(3RT)`, `siginfo.h(3HEAD)`, `signal.h(3HEAD)`, `time.h(3HEAD)`, `attributes(5)`, `standards(5)`

Name timer_create – create a timer

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clock_id, struct sigevent *restrict evp,
     timer_t *restrict timerid);
```

Description The `timer_create()` function creates a timer using the specified clock, *clock_id*, as the timing base. The `timer_create()` function returns, in the location referenced by *timerid*, a timer ID of type `timer_t` used to identify the timer in timer requests. This timer ID will be unique within the calling process until the timer is deleted. The particular clock, *clock_id*, is defined in `<time.h>`. The timer whose ID is returned will be in a disarmed state upon return from `timer_create()`.

The *evp* argument, if non-null, points to a `sigevent` structure. This structure, allocated by the application, defines the asynchronous notification that will occur when the timer expires (see `signal.h(3HEAD)` for event notification details). If the *evp* argument is `NULL`, the effect is as if the *evp* argument pointed to a `sigevent` structure with the `sigev_notify` member having the value `SIGEV_SIGNAL`, the `sigev_signo` having a default signal number, and the `sigev_value` member having the value of the timer ID, *timerid*.

The system defines a set of clocks that can be used as timing bases for per-process timers. The following values for *clock_id* are supported:

`CLOCK_REALTIME`	wall clock
`CLOCK_VIRTUAL`	user CPU usage clock
`CLOCK_PROF`	user and system CPU usage clock
`CLOCK_HIGHRES`	non-adjustable, high-resolution clock

For timers created with a *clock_id* of `CLOCK_HIGHRES`, the system will attempt to use an optimal hardware source. This may include, but is not limited to, per-CPU timer sources. The actual hardware source used is transparent to the user and may change over the lifetime of the timer. For example, if the caller that created the timer were to change its processor binding or its processor set, the system may elect to drive the timer with a hardware source that better reflects the new binding. Timers based on a *clock_id* of `CLOCK_HIGHRES` are ideally suited for interval timers that have minimal jitter tolerence.

Timers are not inherited by a child process across a `fork(2)` and are disarmed and deleted by a call to one of the `exec` functions (see `exec(2)`).

Return Values Upon successful completion, `timer_create()` returns `0` and updates the location referenced by *timerid* to a `timer_t`, which can be passed to the per-process timer calls. If an error occurs, the function returns −1 and sets `errno` to indicate the error. The value of *timerid* is undefined if an error occurs.

Errors The `timer_create()` function will fail if:

`EAGAIN` The system lacks sufficient signal queuing resources to honor the request, or the calling process has already created all of the timers it is allowed by the system.

`EINVAL` The specified clock ID, *clock_id*, is not defined.

`ENOSYS` The `timer_create()` function is not supported by the system.

`EPERM` The specified clock ID, *clock_id*, is `CLOCK_HIGHRES` and the {`PRIV_PROC_CLOCK_HIGHRES`} is not asserted in the effective set of the calling process.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe with exceptions

See Also `exec`(2), `fork`(2), `time`(2), `clock_settime`(3RT), `signal`(3C), `signal.h`(3HEAD), `timer_delete`(3RT), `timer_settime`(3RT), `attributes`(5), `privileges`(5), `standards`(5)

Name timer_delete – delete a timer

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <time.h>

int timer_delete(timer_t timerid);
```

Description The `timer_delete()` function deletes the specified timer, *timerid*, previously created by the `timer_create(3RT)` function. If the timer is armed when `timer_delete()` is called, the behavior will be as if the timer is automatically disarmed before removal. The disposition of pending signals for the deleted timer is unspecified.

Return Values If successful, the function returns `0`. Otherwise, the function returns `−1` and sets `errno` to indicate the error.

Errors The `timer_delete()` function will fail if:

`EINVAL` The timer ID specified by *timerid* is not a valid timer ID.

`ENOSYS` The `timer_delete()` function is not supported by the system.

Attributes See `attributes(5)` for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	MT-Safe with exceptions

See Also `timer_create(3RT)`, `attributes(5)`, `standards(5)`

Name timer_settime, timer_gettime, timer_getoverrun – per-process timers

Synopsis

```
cc [ flag... ] file... -lrt [ library... ]
#include <time.h>

int timer_settime(timer_t timerid, int flags,
     const struct itimerspec *restrict value,
     struct itimerspec *restrict ovalue);

int timer_gettime(timer_t timerid, struct itimerspec *value);

int timer_getoverrun(timer_t timerid);
```

Description The `timer_settime()` function sets the time until the next expiration of the timer specified by *timerid* from the `it_value` member of the *value* argument and arm the timer if the `it_value` member of *value* is non-zero. If the specified timer was already armed when `timer_settime()` is called, this call resets the time until next expiration to the *value* specified. If the `it_value` member of *value* is 0, the timer is disarmed. The effect of disarming or resetting a timer on pending expiration notifications is unspecified.

If the flag `TIMER_ABSTIME` is not set in the argument *flags*, `timer_settime()` behaves as if the time until next expiration is set to be equal to the interval specified by the `it_value` member of *value*. That is, the timer expires in `it_value` nanoseconds from when the call is made. If the flag `TIMER_ABSTIME` is set in the argument *flags*, `timer_settime()` behaves as if the time until next expiration is set to be equal to the difference between the absolute time specified by the `it_value` member of *value* and the current value of the clock associated with *timerid*. That is, the timer expires when the clock reaches the value specified by the `it_value` member of *value*. If the specified time has already passed, the function succeeds and the expiration notification is made.

The reload value of the timer is set to the value specified by the `it_interval` member of *value*. When a timer is armed with a non-zero `it_interval`, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer will be rounded up to the larger multiple of the resolution. Quantization error will not cause the timer to expire earlier than the rounded time value.

If the argument *ovalue* is not `NULL`, the function `timer_settime()` stores, in the location referenced by *ovalue*, a value representing the previous amount of time before the timer would have expired or 0 if the timer was disarmed, together with the previous timer reload value. The members of *ovalue* are subject to the resolution of the timer, and they are the same values that would be returned by a `timer_gettime()` call at that point in time.

The `timer_gettime()` function stores the amount of time until the specified timer, *timerid*, expires and the reload value of the timer into the space pointed to by the *value* argument. The `it_value` member of this structure contains the amount of time before the timer expires, or 0 if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The `it_interval` member of *value* contains the reload value last set by `timer_settime()`.

Only a single signal will be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal will be queued, and a timer overrun occurs. When a timer expiration signal is delivered to or accepted by a process, the `timer_getoverrun()` function returns the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations that occurred between the time the signal was generated (queued) and when it was delivered or accepted, up to but not including an implementation-dependent maximum of `DELAYTIMER_MAX`. If the number of such extra expirations is greater than or equal to `DELAYTIMER_MAX`, then the overrun count will be set to `DELAYTIMER_MAX`. The value returned by `timer_getoverrun()` applies to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, the meaning of the overrun count returned is undefined.

Return Values

If the `timer_settime()` or `timer_gettime()` functions succeed, `0` is returned. If an error occurs for either of these functions, `−1` is returned, and `errno` is set to indicate the error. If the `timer_getoverrun()` function succeeds, it returns the timer expiration overrun count as explained above.

Errors

The `timer_settime()`, `timer_gettime()` and `timer_getoverrun()` functions will fail if:

`EINVAL` The *timerid* argument does not correspond to a timer returned by `timer_create`(3RT) but not yet deleted by `timer_delete`(3RT).

`ENOSYS` The `timer_settime()`, `timer_gettime()`, and `timer_getoverrun()` functions are not supported by the system. The `timer_settime()` function will fail if:

`EINVAL` A *value* structure specified a nanosecond value less than zero or greater than or equal to 1000 million.

Attributes

See `attributes`(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

See Also

`time.h`(3HEAD), `clock_settime`(3RT), `timer_create`(3RT), `timer_delete`(3RT), `attributes`(5), `standards`(5)