# man pages section 3: Threads and Realtime Library Functions

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-0630-10
February 2000

# Contents

**Preface 15**

aiocancel(3AIO) 21

aio_cancel(3RT) 22

aio_error(3RT) 24

aio_fsync(3RT) 26

aioread(3AIO) 28

aiowrite(3AIO) 28

aio_read(3RT) 31

aio_return(3RT) 34

aio_suspend(3RT) 35

aiowait(3AIO) 37

aio_write(3RT) 39

cancellation(3THR) 42

clock_settime(3RT) 48

clock_gettime(3RT) 48

clock_getres(3RT) 48

cond_init(3THR) 50

cond_wait(3THR) 50

cond_timedwait(3THR) 50

cond_signal(3THR) 50

cond_broadcast(3THR) 50

cond_destroy(3THR) 50

condition(3THR) 54

door_bind(3DOOR) 56

door_unbind(3DOOR) 56

door_call(3DOOR) 59

door_create(3DOOR) 62

door_cred(3DOOR) 64

door_info(3DOOR) 65

door_return(3DOOR) 67

door_revoke(3DOOR) 68

door_server_create(3DOOR) 69

fdatasync(3RT) 72

libthread_db(3THR) 73

lio_listio(3RT) 80

mq_close(3RT) 84

mq_getattr(3RT) 85

mq_notify(3RT) 86

mq_open(3RT) 88

mq_receive(3RT) 92

mq_send(3RT) 94

mq_setattr(3RT) 96

mq_unlink(3RT) 97

mutex(3THR) 98

mutex_init(3THR) 100

mutex_destroy(3THR) 100

mutex_lock(3THR) 100

mutex_trylock(3THR) 100

mutex_unlock(3THR) 100

nanosleep(3RT) 112

proc_service(3PROC) 113

ps_lgetregs(3PROC) 116

ps_lsetregs(3PROC) 116

ps_lgetfpregs(3PROC) 116

ps_lsetfpregs(3PROC) 116

ps_lgetxregsize(3PROC) 116

ps_lgetxregs(3PROC) 116

ps_lsetxregs(3PROC) 116

ps_pglobal_lookup(3PROC) 118

ps_pglobal_sym(3PROC) 118

ps_pread(3PROC) 119

ps_pwrite(3PROC) 119

ps_pdread(3PROC) 119

ps_pdwrite(3PROC) 119

ps_ptread(3PROC) 119

ps_ptwrite(3PROC) 119

ps_pstop(3PROC) 120

ps_pcontinue(3PROC) 120

ps_lstop(3PROC) 120

ps_lcontinue(3PROC) 120

ps_lrolltoaddr(3PROC) 120

ps_kill(3PROC) 120

pthread_atfork(3THR) 122

pthread_attr_getdetachstate(3THR) 124

pthread_attr_setdetachstate(3THR) 124

pthread_attr_getguardsize(3THR) 125

pthread_attr_setguardsize(3THR) 125

pthread_attr_getinheritsched(3THR) 127

pthread_attr_setinheritsched(3THR) 127

pthread_attr_getschedparam(3THR) 129

pthread_attr_setschedparam(3THR) 129

pthread_attr_getschedpolicy(3THR) 130

pthread_attr_setschedpolicy(3THR) 130

pthread_attr_getscope(3THR) 131

pthread_attr_setscope(3THR) 131

pthread_attr_getstackaddr(3THR) 133

pthread_attr_setstackaddr(3THR) 133

pthread_attr_getstacksize(3THR) 134

pthread_attr_setstacksize(3THR) 134

pthread_attr_init(3THR) 135

pthread_attr_destroy(3THR) 135

pthread_cancel(3THR) 137

pthread_cleanup_pop(3THR) 138

pthread_cleanup_push(3THR) 139

pthread_condattr_getpshared(3THR) 140

pthread_condattr_setpshared(3THR) 140

pthread_condattr_init(3THR) 142

pthread_condattr_destroy(3THR) 142

pthread_cond_init(3THR) 144

pthread_cond_destroy(3THR) 144

pthread_cond_signal(3THR) 146

pthread_cond_broadcast(3THR) 146

pthread_cond_wait(3THR) 148

pthread_cond_timedwait(3THR) 148

pthread_create(3THR) 150

pthread_detach(3THR) 154

pthread_equal(3THR) 155

pthread_exit(3THR) 156

pthread_getconcurrency(3THR) 158

pthread_setconcurrency(3THR) 158

pthread_getschedparam(3THR) 160

pthread_setschedparam(3THR) 160

pthread_getspecific(3THR) 162

pthread_setspecific(3THR) 162

pthread_join(3THR) 163

pthread_key_create(3THR) 165

pthread_key_delete(3THR) 167

pthread_kill(3THR) 168

pthread_mutexattr_getprioceiling(3THR) 169

pthread_mutexattr_setprioceiling(3THR) 169

pthread_mutexattr_getprotocol(3THR) 171

pthread_mutexattr_setprotocol(3THR) 171

pthread_mutexattr_getpshared(3THR) 174

pthread_mutexattr_setpshared(3THR) 174

pthread_mutexattr_getrobust_np(3THR) 176

pthread_mutexattr_setrobust_np(3THR) 176

pthread_mutexattr_gettype(3THR) 178

pthread_mutexattr_settype(3THR) 178

pthread_mutexattr_init(3THR) 181

pthread_mutexattr_destroy(3THR) 181

pthread_mutex_consistent_np(3THR) 182

pthread_mutex_getprioceiling(3THR) 184
pthread_mutex_setprioceiling(3THR) 184
pthread_mutex_init(3THR) 186
pthread_mutex_destroy(3THR) 186
pthread_mutex_lock(3THR) 188
pthread_mutex_trylock(3THR) 188
pthread_mutex_unlock(3THR) 188
pthread_once(3THR) 192
pthread_rwlockattr_getpshared(3THR) 193
pthread_rwlockattr_setpshared(3THR) 193
pthread_rwlockattr_init(3THR) 194
pthread_rwlockattr_destroy(3THR) 194
pthread_rwlock_init(3THR) 195
pthread_rwlock_destroy(3THR) 195
pthread_rwlock_rdlock(3THR) 197
pthread_rwlock_tryrdlock(3THR) 197
pthread_rwlock_unlock(3THR) 199
pthread_rwlock_wrlock(3THR) 200
pthread_rwlock_trywrlock(3THR) 200
pthread_self(3THR) 202
pthread_setcancelstate(3THR) 203
pthread_setcanceltype(3THR) 205
pthread_sigmask(3THR) 207
pthread_testcancel(3THR) 212
rwlock(3THR) 214
rwlock_init(3THR) 214
rwlock_destroy(3THR) 214
rw_rdlock(3THR) 214

rw_wrlock(3THR) 214
rw_tryrdlock(3THR) 214
rw_trywrlock(3THR) 214
rw_unlock(3THR) 214
schedctl_init(3SCHED) 217
schedctl_lookup(3SCHED) 217
schedctl_exit(3SCHED) 217
schedctl_start(3SCHED) 217
schedctl_stop(3SCHED) 217
sched_getparam(3RT) 219
sched_get_priority_max(3RT) 220
sched_get_priority_min(3RT) 220
sched_getscheduler(3RT) 221
sched_rr_get_interval(3RT) 222
sched_setparam(3RT) 223
sched_setscheduler(3RT) 226
sched_yield(3RT) 229
semaphore(3THR) 230
sema_init(3THR) 230
sema_destroy(3THR) 230
sema_wait(3THR) 230
sema_trywait(3THR) 230
sema_post(3THR) 230
sem_close(3RT) 234
sem_destroy(3RT) 235
sem_getvalue(3RT) 236
sem_init(3RT) 237
sem_open(3RT) 239

sem_post(3RT) 242
sem_unlink(3RT) 244
sem_wait(3RT) 245
sem_trywait(3RT) 245
shm_open(3RT) 248
shm_unlink(3RT) 251
sigqueue(3RT) 252
sigwaitinfo(3RT) 254
sigtimedwait(3RT) 254
td_init(3THR) 256
td_log(3THR) 257
td_sync_get_info(3THR) 258
td_sync_setstate(3THR) 258
td_sync_waiters(3THR) 258
td_ta_enable_stats(3THR) 262
td_ta_reset_stats(3THR) 262
td_ta_get_stats(3THR) 262
td_ta_event_addr(3THR) 264
td_thr_event_enable(3THR) 264
td_ta_set_event(3THR) 264
td_thr_set_event(3THR) 264
td_ta_clear_event(3THR) 264
td_thr_clear_event(3THR) 264
td_ta_event_getmsg(3THR) 264
td_thr_event_getmsg(3THR) 264
td_event_emptyset(3THR) 264
td_event_fillset(3THR) 264
td_event_addset(3THR) 264

td_event_delset(3THR) 264

td_eventismember(3THR) 264

td_eventisempty(3THR) 264

td_ta_get_nthreads(3THR) 269

td_ta_map_addr2sync(3THR) 270

td_ta_map_id2thr(3THR) 271

td_ta_map_lwp2thr(3THR) 271

td_ta_new(3THR) 272

td_ta_delete(3THR) 272

td_ta_get_ph(3THR) 272

td_ta_setconcurrency(3THR) 274

td_ta_sync_iter(3THR) 275

td_ta_thr_iter(3THR) 275

td_ta_tsd_iter(3THR) 275

td_thr_dbsuspend(3THR) 277

td_thr_dbresume(3THR) 277

td_thr_getgregs(3THR) 279

td_thr_setgregs(3THR) 279

td_thr_getfpregs(3THR) 279

td_thr_setfpregs(3THR) 279

td_thr_getxregsize(3THR) 279

td_thr_getxregs(3THR) 279

td_thr_setxregs(3THR) 279

td_thr_get_info(3THR) 281

td_thr_lockowner(3THR) 285

td_thr_setprio(3THR) 286

td_thr_setsigpending(3THR) 287

td_thr_sigsetmask(3THR) 287

td_thr_sleepinfo(3THR) 289

td_thr_tsd(3THR) 290

td_thr_validate(3THR) 291

thr_create(3THR) 292

threads(3THR) 298

pthreads(3THR) 298

libpthread(3THR) 298

libthread(3THR) 298

thr_exit(3THR) 307

thr_getconcurrency(3THR) 309

thr_setconcurrency(3THR) 309

thr_getprio(3THR) 310

thr_setprio(3THR) 310

thr_join(3THR) 312

thr_keycreate(3THR) 314

thr_setspecific(3THR) 314

thr_getspecific(3THR) 314

thr_kill(3THR) 317

thr_main(3THR) 318

thr_min_stack(3THR) 319

thr_self(3THR) 321

thr_sigsetmask(3THR) 322

thr_stksegment(3THR) 327

thr_suspend(3THR) 328

thr_continue(3THR) 328

thr_yield(3THR) 329

timer_create(3RT) 330

timer_delete(3RT) 332

timer_settime(3RT) 333

timer_gettime(3RT) 333

timer_getoverrun(3RT) 333

**Index 335**

# 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. |

CHAPTER

# Introduction to 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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

aio_read(3RT), aio_return(3RT), attributes(5), aio(3HEAD), lf64(5), signal(3HEAD)

**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;
sigsetempty(&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);
 . . .
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 |
|---|---|
| MT-Level | Async-Signal-Safe |

**SEE ALSO**

aio_read(3RT), aio_write(3RT), aio_fsync(3RT), lio_listio(3RT), aio_return(3RT), aio_cancel(3RT), _exit(2), close(2), fork(2), lseek(2), read(2), write(2), attributes(5), aio(3HEAD), lf64(5), signal(3HEAD)

**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 |
|---|---|
| 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(3HEAD), aio(3HEAD), lf64(5), signal(3HEAD)

**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**

`aioread( )` 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* .

`aiowrite( )` 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. `AIO_INPROGRESS` is not a value used by the system so the client may 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 may 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`(3AIO) returns nothing (using a polling timeout) before returning from the signal handler, no asynchronous I/O notifications are lost. The `aiowait`(3AIO) function is the only way to dequeue an asynchronous notification. Note: `SIGIO` may have several meanings simultaneously: for example, 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.

`close`(2) , `exit`(2) and `execve( )` (see `exec`(2) ) will 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 may only be reused 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**

`aioread( )` and `aiowrite( )` will fail if any of the following are true:

| | |
|---|---|
| `EAGAIN` | The number of asynchronous requests that the system can handle at any one time has been exceeded |
| `EBADF` | *fildes* is not a valid file descriptor open for reading. |
| `EFAULT` | At least one of *bufp* points to an address outside the address space of the requesting process. See `NOTES` . |
| `EINVAL` | The parameter *resultp* is currently being used by an outstanding asynchronous request. |
| `EINVAL` | *offset* 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) , exec(2) , exit(2) , llseek(2) , lseek(2) , open(2) , read(2) , write(2) , aiocancel(3AIO) , aiowait(3AIO) , sigvec(3UCB) , attributes(5) , lf64(5)

**NOTES**

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

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

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

**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 |
|---|---|
| MT-Level | Async-Signal-Safe |

**SEE ALSO**

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

**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. 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_suspen( ) function. Note that, since each asynchronous I/O operation 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.

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 |
|---|---|
| MT-Level | Async-Signal-Safe |

**SEE ALSO**

`aio_fsync`(3RT), `aio_read`(3RT), `aio_return`(3RT), `aio_write`(3RT), `lio_listio`(3RT), `attributes`(5), `aio`(3HEAD), `lf64`(5), `signal`(3HEAD)

**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**

aiowait( ) suspends the calling process until one of its outstanding asynchronous I/O operations completes. This provides 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, then 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**

aiowait( ) will fail if any of the following are true:

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

EINTR aiowait( ) was interrupted by a signal.

EINVAL There are no outstanding asynchronous I/O requests.

**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**

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

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

**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 |
|---|---|
| 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), attributes(5), aio(3HEAD), lf64(5), signal(3HEAD)

**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**

cancellation – overview of concepts related to POSIX thread cancellation

**DESCRIPTION**

| FUNCTION | ACTION |
|---|---|
| `pthread_cancel` | Cancels thread execution. |
| `pthread_setcancelstate` | Sets the cancellation *state* of a thread. |
| `pthread_setcanceltype` | Sets the cancellation `type` of a thread. |
| `pthread_testcancel` | Creates a cancellation point in the calling thread. |
| `pthread_cleanup_push` | Pushes a cleanup handler routine. |
| `pthread_cleanup_pop` | Pops a cleanup handler routine. |

**Cancellation**

Thread cancellation allows a thread to terminate the execution of any application thread in the process. Cancellation is useful when further operations of one or more threads are undesirable or unnecessary.

An example of a situation that could benefit from using cancellation is an asynchronously-generated cancel condition such as a user requesting to close or exit some running operation. Another example is the completion of a task undertaken by a number of threads, such as solving a maze. While many threads search for the solution, one of the threads might solve the puzzle while the others continue to operate. Since they are serving no purpose at that point, they should all be canceled.

**Planning Steps**

Planning and programming for most cancellations follow this pattern:

1. Identify which threads you want to cancel, and insert `pthread_cancel`(3THR) statements.
2. Identify system-defined cancellation points where a thread that might be canceled could have changed system or program state that should be restored. See the `Cancellation Points` for a list.
3. When a thread changes the system or program state just before a cancellation point, and should restore that state before the thread is canceled, place a cleanup handler before the cancellation point with `pthread_cleanup_push`(3THR). Wherever a thread restores the changed state, pop the cleanup handler from the cleanup stack with `pthread_cleanup_pop`(3THR).
4. Know whether the threads you are canceling call into cancel-unsafe libraries, and disable cancellation with `pthread_setcancelstate`(3THR) before the call into the library. See `Cancellation State` and `Cancel-Safe`.
5. To cancel a thread in a procedure that contains no cancellation points, insert your own cancellation points with `pthread_testcancel`(3THR).

`pthread_testcancel`(3THR) creates cancellation points by testing for pending cancellations and performing those cancellations if they are found. Push and pop cleanup handlers around the cancellation point, if necessary (see Step 3, above).

**Cancellation Points**

The system defines certain points at which cancellation can occur (cancellation points), and you can create additional cancellation points in your application with `pthread_testcancel`(3THR).

The following cancellation points are defined by the system (system-defined cancellation points): `aio_suspend`(3RT), `close(2)`, `creat`(2), `getmsg`(2), `getpmsg`(2), `lockf`(3C), `mq_receive`(3RT), `mq_send`(3RT), `msgrcv`(2), `msgsnd`(2), `msync`(3C), `nanosleep`(3RT), `open`(2), `pause`(2), `poll`(2), `pread`(2), `pthread_cond_timedwait`(3THR), `pthread_cond_wait`(3THR), `pthread_join`(3THR), `pthread_testcancel`(3THR), `putmsg`(2), `putpmsg`(2), `pwrite`(2), `read`(2), `readv`(2), `select`(3C), `sem_wait`(3RT), `sigpause`(3C), `sigwaitinfo`(3RT), `sigsuspend`(2), `sigtimedwait`(3RT), `sigwait`(2), `sleep`(3C), `sync`(2), `system`(3C), `tcdrain`(3C), `usleep`(3C), `wait`(2), `waitid`(2) `waitpid`(2), `wait3`(3C), `write`(2), `writev`(2), and `fcntl`(2), when specifying F_SETLKW as the command

When cancellation is asynchronous, cancellation can occur before, during, or after the execution of the function defined as the cancellation point. When cancellation is deferred (the default case), cancellation occurs before the function defined as the cancellation point executes. See `Cancellation Type` for more information about deferred and asynchronous cancellation.

Choosing where to place cancellation points and understanding how cancellation affects your program depend upon your understanding of both your application and of cancellation mechanics.

Typically, any call that might require a long wait should be a cancellation point. Operations need to check for pending cancellation requests when the operation is about to block indefinitely. This includes threads waiting in `pthread_cond_wait`(3THR) and `pthread_cond_timedwait`(3THR), threads waiting for the termination of another thread in `pthread_join`(3THR), and threads blocked on `sigwait`(2).

A mutex is explicitly *not* a cancellation point and should be held for only the minimal essential time.

Most of the dangers in performing cancellations deal with properly restoring invariants and freeing shared resources. For example, a carelessly canceled thread might leave a mutex in a locked state, leading to a deadlock. Or it might leave a region of memory allocated with no way to identify it and therefore no way to free it.

**Cleanup Handlers**

When a thread is canceled, it should release resources and clean up the state that is shared with other threads. So, whenever a thread that might be canceled changes the state of the system or of the program, be sure to push a cleanup handler with `pthread_cleanup_push`(3THR) before the cancellation point.

When a thread is canceled, all the currently-stacked cleanup handlers are executed in last-in-first-out (LIFO) order. Each handler is run in the scope in which it was pushed. When the last cleanup handler returns, the thread-specific data destructor functions are called. Thread execution terminates when the last destructor function returns.

When, in the normal course of the program, an uncanceled thread restores state that it had previously changed, be sure to pop the cleanup handler (that you had set up where the change took place) using `pthread_cleanup_pop`(3THR). That way, if the thread is canceled later, only currently-changed state will be restored by the handlers that are left in the stack.

Be sure to pop the handler in the same scope in which it was pushed. Also, make sure that each push statement has a matching pop statement, or compiler errors will be generated.

**Cancellation State**

Most programmers will use only the default cancellation state of `PTHREAD_CANCEL_ENABLE,` but can choose to change the state by using `pthread_setcancelstate`(3THR), which determines whether a thread is cancelable at all. With the default *state* of `PTHREAD_CANCEL_ENABLE,` cancellation is enabled, and the thread is cancelable at points determined by its cancellation *type.* See `Cancellation Type`.

If the *state* is `PTHREAD_CANCEL_DISABLE,` cancellation is disabled, and the thread is not cancelable at any point — all cancellation requests to it are held pending.

You might want to disable cancellation before a call to a cancel-unsafe library, restoring the old cancel state when the call returns from the library. See `Cancel-Safe` for explanations of cancel safety.

**Cancellation Type**

A thread's cancellation `type` is set with `pthread_setcanceltype`(3THR), and determines whether the thread can be canceled anywhere in its execution, or only at cancellation points.

With the default `type` of `PTHREAD_CANCEL_DEFERRED,` the thread is cancelable only at cancellation points, and then only when cancellation is enabled.

If the `type` is `PTHREAD_CANCEL_ASYNCHRONOUS,` the thread is cancelable at any point in its execution (assuming, of course, that cancellation is enabled). Try to limit regions of asynchronous cancellation to sequences with no external dependencies that could result in dangling resources or unresolved state conditions. Using asynchronous cancellation is discouraged because of the danger involved in trying to guarantee correct cleanup handling at absolutely every point in the program.

**Cancellation Type/State Table**

| Type | State | |
|---|---|---|
| | Enabled (Default) | Disabled |
| Deferred (Default) | Cancellation occurs when the target thread reaches a cancellation point and a cancel is pending. (Default) | All cancellation requests to the target thread are held pending. |
| Asynchronous | Receipt of a `pthread_cancel`(3T) call causes immediate cancellation. | All cancellation requests to the target thread are held pending; as soon as cancellation is re-enabled, pending cancellations are executed immediately. |

**Cancel-Safe**

With the arrival of POSIX cancellation, the *cancel-safe* level has been added to the list of MT-Safety levels See `Intro`(3). An application or library is cancel-safe whenever it has arranged for cleanup handlers to restore system or program state wherever cancellation can occur. The application or library is specifically *Deferred-cancel-safe* when it is cancel-safe for threads whose cancellation type is `PTHREAD_CANCEL_DEFERRED` See `Cancellation State`. It is specifically *Asynchronous-cancel-safe* when it is cancel-safe for threads whose cancellation type is `PTHREAD_CANCEL_ASYNCHRONOUS.`

Obviously, it is easier to arrange for deferred cancel safety, as this requires system and program state protection only around cancellation points. In general, expect that most applications and libraries are *not* Asynchronous-cancel-safe.

**POSIX Threads Only**

Note: The cancellation functions described in this reference page are available for POSIX threads, only (the Solaris threads interfaces do not provide cancellation functions).

## EXAMPLES

**EXAMPLE 1** The following short C++ example shows the pushing / popping of cancellation handlers, the disabling / enabling of cancellation, the use of `pthread_testcancel()`, and so on. The `free_res()` cancellation handler in this example is a dummy function that simply prints a message, but that would free resources in a real application. The function `f2()` is called from the main thread, and goes deep into its call stack by calling itself recursively.

Before `f2()` starts running, the newly created thread has probably posted a cancellation on the main thread since the main thread calls `thr_yield()` right after creating thread2. Because cancellation was initially disabled in the main thread, through a call to `pthread_setcancelstate()`, the call to `f2()` from `main()` continues and constructs X at each recursive call, even though the main thread has a pending cancellation.

When `f2()` is called for the fifty-first time (when "i == 50"), `f2()` enables cancellation by calling `pthread_setcancelstate()`. It then establishes a cancellation point for itself by calling `pthread_testcancel()`. (Because a cancellation is pending, a call to a cancellation point such as `read`(2) or `write`(2) would also cancel the caller here.)

After the `main()` thread is canceled at the fifty-first iteration, all the cleanup handlers that were pushed are called in sequence; this is indicated by the calls to `free_res()` and the calls to the destructor for *X*. At each level, the C++ runtime calls the destructor for *X* and then the cancellation handler, `free_res()`. The print messages from `free_res()` and *X*'s destructor show the sequence of calls.

At the end, the main thread is joined by thread2. Because the main thread was canceled, its return status from `pthread_join()` is `PTHREAD_CANCELED.` After the status is printed, thread2 returns, killing the process (since it is the last thread in the process).

```
#include <pthread.h>
#include <sched.h>
extern "C" void thr_yield(void);

extern "C" void printf(...);

struct X {
                int x;
                X(int i){x = i; printf("X(%d) constructed.\n", i);}
                ~X(){ printf("X(%d) destroyed.\n", x);}
};

void
free_res(void *i)
{
                printf("Freeing ‘%d‘\n",i);
}

char* f2(int i)
{
                try {
                X dummy(i);
                pthread_cleanup_push(free_res, (void *)i);
                if (i == 50) {
                       pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
                    pthread_testcancel();
                }
                f2(i+1);
                pthread_cleanup_pop(0);
                }
                catch (int) {
                printf("Error: In handler.\n");
```

```
                }
                return "f2";
}

void *
thread2(void *tid)
{
                void *sts;

                printf("I am new thread :%d\n", pthread_self());

                pthread_cancel((pthread_t)tid);

                pthread_join((pthread_t)tid, &sts);

                printf("main thread cancelled due to %d\n", sts);

                return (sts);
}

main()
{
                pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
                pthread_create(NULL, NULL, thread2, (void *)pthread_self());
                thr_yield();
                printf("Returned from %s\n",f2(0));
}
```

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

read(2), sigwait(2), write(2), Intro(3), condition(3THR), pthread_cleanup_pop(3THR), pthread_cleanup_push(3THR), pthread_exit(3THR), pthread_join(3THR), pthread_setcancelstate(3THR), pthread_setcanceltype(3THR), pthread_testcancel(3THR), setjmp(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 |
|---|---|
| MT-Level | `clock_gettime()` is Async-Signal-Safe |

**SEE ALSO**

`time`(2), `ctime`(3C), `gethrtime`(3C), `time`(3HEAD), `timer_gettime`(3RT), `attributes`(5)

**NAME**

cond_init, cond_wait, cond_timedwait, cond_signal, cond_broadcast, cond_destroy – condition variables

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...[ *library*... ]

```
#include &lt;thread.h&gt;
#include &lt;synch.h&gt;
int cond_init(cond_t *cvp, int type, void *arg);

int cond_wait(cond_t *cvp, mutex_t *mp);

int cond_timedwait(cond_t *cvp, mutex_t *mp, timestruc_t *abstime);

int cond_signal(cond_t *cvp);

int cond_broadcast(cond_t *cvp);

int cond_destroy(cond_t *cvp);
```

**DESCRIPTION**

**Initialize**

Condition variables and mutexes should be global. Condition variables that are allocated in writable memory can synchronize threads among processes if they are shared by the cooperating processes (see mmap(2) ) and are initialized for this purpose.

The scope of a condition variable is either intra-process or inter-process. This is dependent upon whether the argument is passed implicitly or explicitly to the initialization of that condition variable. A condition variable does not need to be explicitly initialized. A condition variable is initialized with all zeros, by default, and its scope is set to within the calling process. For inter-process synchronization, a condition variable must be initialized once, and only once, before use.

A condition variable must not be simultaneously initialized by multiple threads or re-initialized while in use by other threads.

Condition variables' attributes may be set to the default or customized at initialization.

cond_init( ) initializes the condition variable pointed to by *cvp* . A condition variable can have several different types of behavior, specified by type . No current type uses *arg* although a future type may specify additional behavior parameters via *arg* . type may be one of the following:

USYNC_THREAD — The condition variable can synchronize threads only in this process. This is the default.

USYNC_PROCESS — The condition variable can synchronize threads in this process and other processes. Only one process should initialize the condition variable. The object initialized with this attribute must be allocated in memory shared between processes, either in System V shared memory (see `shmop`(2) ) or in memory mapped to a file (see `mmap`(2) ). It is illegal to initialize the object this way and to not allocate it in such shared memory.

Initializing condition variables can also be accomplished by allocating in zeroed memory, in which case, a `type` of `USYNC_THREAD` is assumed.

If default condition variable attributes are used, statically allocated condition variables can be initialized by the macro `DEFAULTCV.`

Default condition variable initialization (intra-process):

```
cond_t cvp;

cond_init(&cvp, NULL, NULL); /* initialize condition variable with default */ OR

cond_init(&cvp, USYNC_THREAD, NULL); OR

cond_t  cond  = DEFAULTCV;
```

Customized condition variable initialization (inter-process):

```
cond_init(&cvp, USYNC_PROCESS, NULL); /* initialize cv with inter-process scope */
```

**Condition Wait**

The condition wait interface allows a thread to wait for a condition and atomically release the associated mutex that it needs to hold to check the condition. The thread waits for another thread to make the condition true and that thread's resulting call to signal and wakeup the waiting thread.

`cond_wait( )` atomically releases the mutex pointed to by *mp* and causes the calling thread to block on the condition variable pointed to by *cvp* . The blocked thread may be awakened by `cond_signal( )` , `cond_broadcast( )` , or when interrupted by delivery of a UNIX signal or a `fork( )` .

`cond_wait( )` and `cond_timedwait( )` always return with the mutex locked and owned by the calling thread even when returning an error.

**Condition Signaling**

A condition signal allows a thread to unblock the next thread waiting on the condition variable, whereas, a condition broadcast allows a thread to unblock all threads waiting on the condition variable.

`cond_signal( )` unblocks one thread that is blocked on the condition variable pointed to by *cvp* .

`cond_broadcast( )` unblocks all threads that are blocked on the condition variable pointed to by *cvp* .

If no threads are blocked on the condition variable, then `cond_signal()` and `cond_broadcast()` have no effect.

Both functions should be called under the protection of the same mutex that is used with the condition variable being signaled. Otherwise, the condition variable may be signaled between the test of the associated condition and blocking in `cond_wait()`. This can cause an infinite wait.

**Destroy**

The condition destroy functions destroy any state, but not the space, associated with the condition variable.

`cond_destroy()` destroys any state associated with the condition variable pointed to by *cvp* . The space for storing the condition variable is not freed.

## RETURN VALUES

Upon successful completion, these functions return `0` . Otherwise, a non-zero value is returned to indicate the error.

## ERRORS

These functions may fail if:

`EFAULT` *cond* , *attr* , *cvp* , *arg* , *abstime* , or *mutex* point to an illegal address.

`EINVAL` Invalid argument. For `cond_init()` , `type` is not a recognized type. For `cond_timedwait()` , the specified number of seconds, *abstime* , is greater than *current_time* + `100,000,000` , where *current_time* is the current time, or the number of nanoseconds is greater than or equal to `1,000,000,000` .

The `cond_timedwait()` function may fail if:

`ETIME` The time specified by *abstime* has passed.

## EXAMPLES

**EXAMPLE 1** `cond_wait()` is normally used in a loop testing some condition, as follows:

```
(void) mutex_lock(mp);
 while (cond == FALSE) {
      (void) cond_wait(cvp, mp);
 }
 (void) mutex_unlock(mp);
```

**EXAMPLE 2** `cond_timedwait()` is also normally used in a loop testing in some conditions. It uses an absolute timeout value as follows:

```
timestruc_t to;
 ...
 (void) mutex_lock(mp);
 to.tv_sec = time(NULL) + TIMEOUT;
 to.tv_nsec = 0;
 while (cond == FALSE) {
      err = cond_timedwait(cvp, mp, &to);
      if (err == ETIMEDOUT) {
           /* timeout, do something */
```

```
                break;
            }
        }
        (void) mutex_unlock(mp);
```

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

fork(2) , mmap(2) , setitimer(2) , shmop(2) , condition(3THR) , mutex(3THR) , signal(3C) , attributes(5) , standards(5)

**NOTES**

The only policy currently supported is SCHED_OTHER . In Solaris, under the SCHED_OTHER policy, there is no established order in which threads are unblocked.

If more than one thread is blocked on a condition variable, the order in which threads are unblocked is determined by the scheduling policy. When each thread, unblocked as a result of a cond_signal( ) or cond_broadcast( ) , returns from its call to cond_wait( ) or cond_timedwait( ) , the thread owns the mutex with which it called cond_wait( ) or cond_timedwait( ) . The thread(s) that are unblocked compete for the mutex according to the scheduling policy, and as if each had called mutex_lock(3THR) .

When cond_wait( ) returns the value of the condition is indeterminate and must be reevaluated.

cond_timedwait( ) is similar to cond_wait( ) , except that the calling thread will not wait for the condition to become true past the absolute time specified by *abstime* . Note that cond_timedwait( ) may continue to block as it trys to reacquire the mutex pointed to by *mp* , which may be locked by another thread. If *abstime* then cond_timedwait( ) returns because of a timeout, it returns the error code ETIME .

**NAME**

condition – concepts related to condition variables

**DESCRIPTION**

Occasionally, a thread running within a mutex needs to wait for an event, in which case it blocks or sleeps. When a thread is waiting for another thread to communicate its disposition, it uses a condition variable in conjunction with a mutex. Although a mutex is exclusive and the code it protects is sharable (at certain moments), condition variables enable the synchronization of differing events that share a mutex, but not necessarily data. Several condition variables may be used by threads to signal each other when a task is complete, which then allows the next waiting thread to take ownership of the mutex.

A condition variable enables threads to atomically block and test the condition under the protection of a mutual exclusion lock (mutex) until the condition is satisfied. If the condition is false, a thread blocks on a condition variable and atomically releases the mutex that is waiting for the condition to change. If another thread changes the condition, it may wake up waiting threads by signaling the associated condition variable. The waiting threads, upon awakening, reacquire the mutex and re-evaluate the condition.

**Initialize**

Condition variables and mutexes should be global. Condition variables that are allocated in writable memory can synchronize threads among processes if they are shared by the cooperating processes (see `mmap`(2)) and are initialized for this purpose.

The scope of a condition variable is either intra-process or inter-process. This is dependent upon whether the argument is passed implicitly or explicitly to the initialization of that condition variable. A condition variable does not need to be explicitly initialized. A condition variable is initialized with all zeros, by default, and its scope is set to within the calling process. For inter-process synchronization, a condition variable must be initialized once, and only once, before use.

A condition variable must not be simultaneously initialized by multiple threads or re-initialized while in use by other threads.

Condition variables attributes may be set to the default or customized at initialization. POSIX threads even allow the default values to be customized. Establishing these attributes varies depending upon whether POSIX or Solaris threads are used. Similar to the distinctions between POSIX and Solaris thread creation, POSIX condition variables implement the default, intra-process, unless an attribute object is modified for inter-process prior to the initialization of the condition variable. Solaris condition variables also implement as the default, intra-process; however, they set this attribute according to the argument, *type*, passed to their initialization function.

**Condition Wait**

The condition wait interface allows a thread to wait for a condition and atomically release the associated mutex that it needs to hold to check the condition. The thread waits for another thread to make the condition true and that thread's resulting call to signal and wakeup the waiting thread.

**Condition Signaling**

A condition signal allows a thread to unblock the next thread waiting on the condition variable, whereas, a condition broadcast allows a thread to unblock all threads waiting on the condition variable.

**Destroy**

The condition destroy functions destroy any state, but not the space, associated with the condition variable.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`fork`(2), `mmap`(2), `setitimer`(2), `shmop`(2), `cond_init`(3THR), `cond_wait`(3THR), `cond_timedwait`(3THR), `cond_signal`(3THR), `cond_broadcast`(3THR), `cond_destroy`(3THR), `mutex`(3THR), `pthread_condattr_init`(3THR), `pthread_cond_init`(3THR), `pthread_cond_wait`(3THR), `pthread_cond_timedwait`(3THR), `pthread_cond_signal`(3THR), `pthread_cond_broadcast`(3THR), `pthread_cond_destroy`(3THR), `signal`(3C), `attributes`(5), `standards`(5)

## NOTES

If more than one thread is blocked on a condition variable, the order in which threads are unblocked is determined by the scheduling policy.

`USYNC_THREAD` does not support multiple mapplings to the same logical synch object. If you need to `mmap()` a synch object to different locations within the same address space, then the synch object should be initialized as a shared object `USYNC_PROCESS` for Solaris, and `PTHREAD_PROCESS_PRIVATE` for POSIX.

**NAME**

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

**SYNOPSIS**

cc [ *flag* ... ] *file* ... –ldoor –lthread [ *library* ... ]

#include <door.h>

int **door_bind**(int*did*);

int **door_unbind**();

**DESCRIPTION**

door_bind( ) 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* .

door_unbind( ) 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( ) . Therefore, it is necessary to bind at least one server thread to doors created with DOOR_PRIVATE.

The server thread create routine, 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. Upon failure, a -1 is returned and errno is set to indicate the error.

**ERRORS**

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

| | |
|---|---|
| EBADF | *did* is not a valid door |
| EBADF | door_unbind( ) with a server thread that is currently not bound |
| EINVAL | *did* was not created with the DOOR_PRIVATE attribute |

## EXAMPLES

**EXAMPLE 1** Using `door_bind( )`

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 foo(); extern bar();

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(POSIX_CANCEL_DISABLE, NULL);
        door_return(NULL, 0, NULL, 0); /* Wait for door invocation */
}

static void
my_create(door_info_t *dip)

        /* 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);
        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 |
| Stability | Evolving |
| MT-Level | Safe |

**SEE ALSO**

`fork`(2) , `door_create`(3DOOR) , `door_return`(3DOOR) , `door_server_create`(3DOOR) , `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 may be passed or returned is a file descriptor, denoted by the `DOOR_DESCRIPTOR` attribute. Addiionally, the `DOOR_RELEASE` attribute may be set, which will cause 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_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`(3THR).

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()`. Programs 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:

| | |
|---|---|
| `EBADF` | Invalid door descriptor was passed |
| `EINVAL` | Bad arguments were passed |
| `EFAULT` | Argument pointers pointed outside the allocated address space |
| `E2BIG` | Arguments were too big for server thread stack |
| `EOVERFLOW` | System could not create overflow area in caller for results. |
| `EAGAIN` | Server was out of available resources |
| `EINTR` | Signal was caught in the client during the invocation |
| `EMFILE` | The client or server has too many open descriptors |

**ATTRIBUTES**

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

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

**SEE ALSO**

`munmap`(2), `cancellation`(3THR), `door_create`(3DOOR), `door_info`(3DOOR), `door_return`(3DOOR), `attributes`(5), `standards`(5)

**NAME**

door_create – create a door descriptor

**SYNOPSIS**

cc [ *flag* ... ] *file* ... –ldoor –lthread [ *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*. *argp* points to *arg_size* bytes of data and *dp* points to *n_desc* door_desc_t structures. The *attributes* flag specifies attributes associated with the newly created door. Valid values for *attributes* are constructed by OR-ing in 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).

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). Programs 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.

**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.

**ATTRIBUTES**

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

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

**SEE ALSO**

door_bind(3DOOR), door_call(3DOOR), door_info(3DOOR), door_revoke(3DOOR), door_server_create(3DOOR), attributes(5)

**NAME**

door_cred – return credential information associated with the client

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–ldoor –lthread` [ *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`. Upon failure, `door_cred( )` returns `–1` and sets `errno` to indicate the error.

**ERRORS**

The `door_cred( )` function fails if:

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

`EINVAL` There is no associated door client.

**ATTRIBUTES**

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

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

**SEE ALSO**

`door_call`(3DOOR), `door_create`(3DOOR), `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 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_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.

**SEE ALSO**

`door_bind`(3DOOR), `door_create`(3DOOR), `door_server_create`(3DOOR)

**NAME**

door_return – return from a door invocation

**SYNOPSIS**

```
cc [ flag ... ] file ... -ldoor -lthread [ 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_prt* and *desc_ptr*. If there is not a client associated with the `door_return( )`, the calling thread discards the results and blocks waiting for the next door invocation.

**RETURN VALUES**

Upon successful completion, `door_return( )` does not return to the calling process. Upon failure, `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_prt* 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. |

**SEE ALSO**

`door_call`(3DOOR)

**NAME**

door_revoke – revoke access to a door descriptor

**SYNOPSIS**

```
cc [ flag ... ] file ... -ldoor -lthread [ 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). `door_revoke( )` 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`. Upon failure, `door_revoke( )` returns `–1` and sets `errno` to indicate the error.

**ERRORS**

The `door_revoke( )` function fails 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 |
| Stability | Evolving |
| MT-Level | Safe |

**SEE ALSO**

`close`(2), `door_create`(3DOOR), `attributes`(5)

**NAME**

door_server_create – specify an alternative door server thread creation function

**SYNOPSIS**

cc [ *flag* ... ] *file* ... −ldoor −lthread [ *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 may 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(3THR).

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 |
| Stability | Evolving |
| MT-Level | Safe |

**SEE ALSO**

`cancellation`(3THR), `door_bind`(3DOOR), `door_call`(3DOOR), `door_create`(3DOOR), `door_return`(3DOOR), `pthread_create` (3THR), pthread_setcancelstate(3THR), thr_create(3THR), attributes(5), standards(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 |
|---|---|
| MT-Level | Async-Signal-Safe |

**SEE ALSO**

`fcntl`(2), `open`(2), `read`(2), `write`(2), `fsync`(3C), `aio_fsync`(3RT), `attributes`(5), `fcntl`(3HEAD)

**NAME**

libthread_db – library of interfaces for monitoring and manipulating threads-related aspects of multithreaded programs

**SYNOPSIS**

cc [ *flag* ... ] *file* ... `-lthread_db` [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>
void **td_event_addset**(td_thr_events_t *, td_thr_events_e *n*);

void **td_event_delset**(td_thr_events_t *, td_thr_events_e *n*);

void **td_event_emptyset**(td_thr_events_t *);

void **td_event_fillset**(td_thr_events_t *);

void **td_eventisempty**(td_thr_events_t *);

void **td_eventismember**(td_thr_events_t *, td_thr_events_e *n*);

td_err_e **td_init**();

void **td_log**();

td_err_e **td_sync_get_info**(const td_synchandle_t **sh_p*, td_syncinfo_t **si_p*);

td_err_e **td_sync_setstate**(const td_synchandle_t **sh_p*, int *value*);

td_err_e **td_sync_waiters**(const td_synchandle_t **sh_p*, td_thr_iter_f **cb*, void **cb_data_p*);

td_err_e **td_thr_clear_event**(const td_thrhandle_t **th_p*, td_thr_events_t **events*);

td_err_e **td_ta_delete**(td_thragent_t **ta_p*);

td_err_e td **ta_enable_stats**(const td_thragent_t**ta_p*, int *on_off*);

td_err_e **td_ta_event_addr**(const td_thragent_t**ta_p*, u_long event, td_notify_t **notify_p*);

td_err_e **td_ta_event_getmsg**(const td_thragent_t **ta_p*, td_event_msg_t **msg*);

td_err_e **td_ta_get_nthreads**(const td_thragent_t **ta_p*, int **nthread_p*);

td_err_e **td_ta_get_ph**(const td_thragent_t **ta_p*, struct ps_prochandle ***ph_pp*);

td_err_e **td_ta_get_stats**(const td_thragent_t **ta_p*, td_ta_stats_t **tstats*);

td_err_e **td_ta_map_addr2sync**(const td_thragent_t **ta_p*, psaddr_t addr td_synchandle_t **sh_p*);

td_err_e **td_ta_map_id2thr**(const td_thragent_t **ta_p*, thread_t tid, td_thrhandle_t **th_p*);

td_err_e **td_ta_map_lwp2thr**(const td_thragent_t **ta_p*, lwpid_t lwpid, td_thrhandle_t **th_p*);

td_err_e **td_ta_new**(struct ps_prochandle **ph_p*, td_thragent_t ***ta_pp*);

td_err_e **td_ta_reset_stats**(const td_thragent_t **ta_p*);

td_err_e **td_ta_setconcurrency**(const td_thragent_t **ta_p*, int *level*);

td_err_e **td_ta_sync_iter**(const td_thragent_t **ta_p*, td_sync_iter_f **cb*, void **cbdata_p*);

td_err_e **td_ta_thr_iter**(const td_thragent_t **ta_p*, td_key_iter_f **cb*, void **cbdata_p*);

td_err_e **td_ta_tsd_iter**(const td_thragent_t **ta_p*, td_key_iter_f **cb*, void **cbdata_p*);

td_err_e **td_thr_clear_event**(const td_thrhandle_t **th_p*, td_thr_events_t **events*);

td_err_e **td_thr_dbresume**(const td_thrhandle_t **th_p*);

td_err_e **td_thr_dbsuspend**(const td_thrhandle_t **th_p*);

td_err_e **td_thr_event_enable**(const td_thrhandle_t **th_p*, int *on_off*);

td_err_e **td_thr_event_getmsg**(const td_thrhandle_t, td_event_msg_t **msg*);

td_err_e **td_thr_get_info**(const td_thrhandle_t **th_p*, td_thrinfo_t **ti_p*);

td_err_e **td_thr_getfpregs**(const td_thrhandle_t **th_p*, prfpregset_t **fpregset*);

td_err_e **td_thr_getgregs**(const td_thrhandle_t **th_p*, prgregset_t *regset*);

td_err_e **td_thr_getxregs**(const td_thrhandle_t **th_p*, void **xregset*);

td_err_e **td_thr_getxregsize**(const td_thrhandle_t **th_p*, int **xregsize*);

td_err_e **td_thr_lockowner**(const td_thrhandle_t **th_p*, td_sync_iter_f **cb*, void **cb_data_p*);

td_err_e **td_thr_set_event**(const td_thrhandle_t **th_p*, td_thr_events_t **events*);

td_err_e **td_thr_setfpregs**(const td_thrhandle_t **th_p*, prfpregset_t **fpregset*);

td_err_e **td_thr_setgregs**(const td_thrhandle_t **th_p*, const prgregset_t *regset*);

td_err_e **td_thr_setprio**(const td_thrhandle_t **th_p*, const int *new_prio*);

td_err_e **td_thr_setsigpending**(const td_thrhandle_t **th_p*, const uchar_t, ti_pending_flag, const sigset_t *ti_pending*);

td_err_e **td_thr_setxregs**(const td_thrhandle_t **th_p*, const void **xregset*);

td_err_e **td_thr_sigsetmask**(const td_thrhandle_t **th_p*, const sigset_t *ti_sigmask*);

td_err_e **td_thr_sleepinfo**(const td_thrhandle_t **th_p*, td_synchandle_t **sh_p*);

td_err_e **td_thr_tsd**(const td_thrhandle_t **th_p*, const thread_key_t key, void ***data_pp*);

td_err_e **td_thr_validate**(const td_thrhandle_t **th_p*);

**DESCRIPTION**

libthread_db is a library that provides support for monitoring and manipulating threads-related aspects of a multithreaded program. There are at least two processes involved, the controlling process and one or more target processes. The controlling process is the libthread_db client, which links with libthread_db and uses libthread_db to inspect or modify threads-related aspects of one or more target processes. The target processes must be multithreaded processes that use libthread or libpthread. The controlling process may or may not be multithreaded itself.

The most commonly anticipated use for libthread_db is that the controlling process will be a debugger for a multithreaded program, hence the "db" in libthread_db.

libthread_db is dependent on the internal implementation details of libthread. It is a "friend" of libthread in the C++ sense, which is precisely the "value added" by libthread_db. It encapsulates the knowledge of libthread internals that a debugger needs in order to manipulate the threads-related state of a target process.

To be able to inspect and manipulate target processes, libthread_db makes use of certain process control primitives that must be provided by the process using libthread_db. The imported interfaces are defined in proc_service(3PROC). In other words, the controlling process is linked with libthread_db, and it calls routines in libthread_db. libthread_db in turn calls certain routines that it expects the controlling process to provide. These process control primitives allow libthread_db to:

- Look up symbols in a target process.
- Stop and continue individual lightweight processes ( LWPs) within a target process.
- Stop and continue an entire target process.
- Read and write memory and registers in a target process.

Initially, a controlling process obtains a handle for a target process. Through that handle it can then obtain handles for the component objects of the target process, its threads, its synchronization objects, and its thread-specific-data keys.

When libthread_db needs to return sets of handles to the controlling process, for example, when returning handles for all the threads in a target process, it uses an iterator function. An iterator function calls back a client-specified function once for each handle to be returned, passing one handle back on each call to the callback function. The calling function also passes another parameter to the iterator function, which the iterator function passes on to the callback function. This makes it easy to build a linked list of thread handles for a particular target process. The additional parameter is the head of the linked list, and the callback function simply inserts the current handle into the linked list.

Callback functions are expected to return an integer. Iteration terminates early if a callback function returns a non-zero value. Otherwise, iteration terminates when there are no more handles to pass back.

`libthread_db` relies on an "agent thread" in the target process for some of its operations. The "agent thread" is a system thread started when `libthread_db` attaches to a process through `td_ta_new`(3THR). In the current implementation, a brief window exists after the agent thread has been started, but before it has completed its initialization, in which `libthread_db` routines that require the agent thread will fail, returning a `TD_NOCAPAB` error status. This is particularly troublesome if the target process was stopped when `td_ta_new()` was called, so that the agent thread cannot be initialized. To avoid this problem, the target process must be allowed to make some forward progress after `td_ta_new()` is called. This limitation will be removed in a future release.

## FUNCTIONS

| Name | Description |
|---|---|
| `td_event_addset()` | Macro that adds a specific event type to an event set. |
| `td_event_delset()` | Macro that deletes a specific event type from an event set. |
| `td_event_emptyset()` | Macro that sets argument to NULL event set. |
| `td_event_fillset()` | Macro that sets argument to set of all events. |
| `td_eventisempty()` | Macro that tests whether an event set is the NULL set. |
| `td_eventismember()` | Macro that tests whether a specific event type is a member of an event set. |
| `td_init()` | Performs initialization for interfaces. |
| `td_log()` | Placeholder for future logging functionality. |
| `td_sync_get_info()` | Gets information for the synchronization object. |
| `td_sync_setstate()` | Sets the state of the synchronization object. |

| | |
|---|---|
| `td_sync_waiters()` | Iteration function used for return of synchronization object handles. |
| `td_ta_clear_event()` | Clears a set of event types in the process event mask. |
| `td_ta_delete()` | Deregisters target process and deallocates internal process handle. |
| `td_ta_enable_stats()` | Turns statistics gathering on or off for the target process. |
| `td_ta_event_addr()` | Returns event reporting address. |
| `td_ta_event_getmsg()` | Returns process event message. |
| `td_ta_get_nthreads()` | Gets the total number of threads in a process. . |
| `td_ta_get_ph()` | Returns corresponding external process handle. |
| `td_ta_get_stats()` | Gets statistics gathered for the target process. |
| `td_ta_map_addr2sync()` | Gets a synchronization object handles from a synchronization object's address. |
| `td_ta_map_id2thr()` | Returns a thread handle for the given thread id. |
| `td_ta_map_lwp2thr()` | Returns a thread handle for the given LWP id. |
| `td_ta_new()` | Registers target process and allocates internal process handle. |
| `td_ta_reset_stats()` | Resets all counters for statistics gathering for the target process. |
| `td_ta_setconcurrency()` | Sets concurrency level for target process. |
| `td_ta_set_event()` | Sets a set of event types in the process event mask. |
| `td_ta_sync_iter()` | Returns handles of synchronization objects associated with a process. |
| `td_ta_thr_iter()` | Returns handles for threads that are part of the target process. |

| | |
|---|---|
| `td_ta_tsd_iter()` | Returns the thread-specific data keys in use by the current process. |
| `td_thr_clear_event()` | Clears a set of event types in the threads event mask. |
| `td_thr_dbresume()` | Resumes thread. |
| `td_thr_dbsuspend()` | Suspends thread. |
| `td_thr_event_enable()` | Enables or disables event reporting. |
| `td_thr_event_getmsg()` | Returns a process event message. |
| `td_thr_get_info()` | Gets thread information and updates |
| `td_thr_getfpregs()` | Gets the floating point registers for the given thread. |
| `td_thr_getgregs()` | Gets the general registers for a given thread. |
| `td_thr_getxregs()` | Gets the extra registers for the given thread. |
| `td_thr_getxregsize()` | Gets the size of the extra register set for the given thread. |
| `td_thr_lockowner()` | Iterates over the set of locks owned by a thread. `struct`. |
| `td_thr_set_event()` | Sets a set of event types in the threads event mask. |
| `td_thr_setfpregs()` | Sets the floating point registers for the given thread. *ti_sigmask* |
| `td_thr_setgregs()` | Sets the general registers for a given thread. |
| `td_thr_setprio()` | Sets the priority of a thread. |
| `td_thr_setsigpending()` | Changes a thread’s pending signal state. |
| `td_thr_setxregs()` | Sets the extra registers for the given thread. |
| `td_thr_sigsetmask()` | Sets the signal mask of the thread. |
| `td_thr_sleepinfo()` | Returns the synchronization handle for the object on which a thread is blocked. |

| | |
|---|---|
| `td_thr_tsd()` | Gets a thread's thread-specific data. |
| `td_thr_validate()` | Tests a thread handle for validity. |

**FILES**

`lthread_db`

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT Level | Safe |

**SEE ALSO**

`libthread`(3THR), `proc_service`(3PROC), `td_event_addset`(3THR), `td_event_delset`(3THR), `td_event_emptyset`(3THR), `td_event_fillset`(3THR), `td_eventisempty`(3THR), `td_eventismember`(3THR), `td_init`(3THR), `td_log`(3THR), `td_sync_get_info`(3THR), `td_sync_waiters`(3THR), `td_ta_delete`(3THR), `td_ta_enable_stats`(3THR), `td_ta_event_addr`(3THR), `td_ta_event_getmsg`(3THR), `td_ta_get_nthreads`(3THR), `td_ta_get_ph`(3THR), `td_ta_get_stats`(3THR), `td_ta_map_addr2sync`(3THR), `td_ta_map_id2thr`(3THR), `td_ta_map_lwp2thr`(3THR), `td_ta_new`(3THR), `td_ta_reset_stats`(3THR), `td_ta_set_event`(3THR), `td_ta_setconcurrency`(3THR), `td_ta_sync_iter`(3THR), `td_ta_thr_iter`(3THR), `td_ta_tsd_iter`(3THR), `td_thr_clear_event`(3THR), `td_thr_dbresume`(3THR), `td_thr_dbsuspend`(3THR), `td_thr_event_enable`(3THR), `td_thr_event_getmsg`(3THR), `td_thr_get_info`(3THR), `td_thr_getfpregs`(3THR), `td_thr_getxregs`(3THR), `td_thr_getxregsize`(3THR), `td_thr_lockowner`(3THR), `td_thr_set_event`(3THR), `td_thr_setfpregs`(3THR), `td_thr_setgregs`(3THR), `td_thr_setprio`(3THR), `td_thr_sigsetmask`(3THR), `td_thr_setsigpending`(3THR), `td_thr_setxregs`(3THR), `td_thr_sleepinfo`(3THR), `td_thr_tsd`(3THR), `td_thr_validate`(3THR), `thr_getspecific`(3THR), `libthread`(3LIB), `libthread_db`(3LIB), `attributes`(5)

**NAME**

lio_listio – list directed I/O

**SYNOPSIS**

`cc` [ *flag*... ] *file*... –`lrt` [ *library*... ]

#include <aio.h>

int **lio_listio**(int *mode*, struct aiocb * const *list*[], int *nent*, struct sigevent **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`(3HEAD)).

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`(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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`close`(2), `exec`(2), `exit`(2), `fork`(2), `lseek`(2), `read`(2), `write`(2), `aio_cancel`(3RT), `aio_fsync`(3RT), `aio_read`(3RT), `aio_return`(3RT), `attributes`(5), `aio`(3HEAD), `fcntl`(3HEAD), `lf64`(5), `siginfo`(3HEAD), `signal`(3HEAD)

**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 &lt;mqueue.h&gt;
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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`mq_notify`(3RT), `mq_open`(3RT), `mq_unlink`(3RT), `attributes`(5), `mqueue`(3HEAD)

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`msgctl`(2), `msgget`(2), `msgrcv`(2), `msgsnd`(2), `mq_open`(3RT), `mq_send`(3RT), `mq_setattr`(3RT), `attributes`(5), `mqueue`(3HEAD)

**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`(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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

mq_close(3RT), mq_open(3RT), mq_receive(3RT), mq_send(3RT), attributes(5), mqueue(3HEAD), siginfo(3HEAD), signal(3HEAD)

**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:

| | |
|---|---|
| EACCESS | 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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

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

**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 – 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 int **msg_prio*);

**DESCRIPTION**

The mq_receive( ) function is used to receive 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 *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.

**RETURN VALUES**

Upon successful completion, mq_receive( ) returns 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 function returns a value of −1, and sets errno to indicate the error condition.

**ERRORS**

The mq_receive( ) function 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.

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

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

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

The mq_receive( ) function 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 |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO**

mq_open(3RT), mq_send(3RT), mq_setattr(3RT), attributes(5), mqueue(3HEAD)

**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_send – 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 int msg_prio);`

**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.

**RETURN VALUES**

Upon successful completion, `mq_send()` returns `0`; otherwise, no message is enqueued, the function returns `–1`, and `errno` is set to indicate the error.

**ERRORS**

The `mq_send()` function 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 call to `mq_send()`

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

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

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

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

mq_open(3RT), mq_receive(3RT), mq_setattr(3RT), sysconf(3C), attributes(5), mqueue(3HEAD)

**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_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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

mq_getattr(3RT), mq_open(3RT), mq_receive(3RT), mq_send(3RT), attributes(5), mqueue(3HEAD)

**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:

| | |
|---|---|
| EACCESS | 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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

mq_close(3RT), mq_open(3RT), attributes(5), mqueue(3HEAD)

**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**

mutex – concepts relating to mutual exclusion locks

**DESCRIPTION**

Mutual exclusion locks (mutexes) prevent multiple threads from simultaneously executing critical sections of code which access shared data (that is, mutexes are used to serialize the execution of threads). All mutexes must be global. A successful call to acquire a mutex will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks the mutex.

Mutexes can synchronize threads within the same process or in other processes. Mutexes can be used to synchronize threads between processes if the mutexes are allocated in writable memory and shared among the cooperating processes (see `mmap`(2)), and have been initialized for this task.

The following table lists mutex functions and the actions they perform.

| FUNCTION | ACTION |
|---|---|
| `mutex_init` | Initialize a mutex. |
| `mutex_destroy` | Destroy a mutex. |
| `mutex_lock` | Lock a mutex. |
| `mutex_trylock` | Attempt to lock a mutex. |
| `mutex_unlock` | Unlock a mutex. |
| `pthread_mutex_init` | Initialize a mutex. |
| `pthread_mutex_destroy` | Destroy a mutex. |
| `pthread_mutex_lock` | Lock a mutex. |
| `pthread_mutex_trylock` | Attempt to lock a mutex. |
| `pthread_mutex_unlock` | Unlock a mutex. |

**Initialization**

Mutexes are either intra-process or inter-process, depending upon the argument passed implicitly or explicitly to the initialization of that mutex. A statically allocated mutex does not need to be explicitly initialized; by default, a statically allocated mutex is initialized with all zeros and its scope is set to be within the calling process.

For inter-process synchronization, a mutex needs to be allocated in memory shared between these processes. Since the memory for such a mutex must be allocated dynamically, the mutex needs to be explicitly initialized with the appropriate attribute that indicates inter-process use.

**Locking and Unlocking**

A critical section of code is enclosed by a call to lock the mutex and the call to unlock the mutex to protect it from simultaneous access by multiple threads. Only one thread at a time may possess mutually exclusive access to the critical section of code that is enclosed by the mutex-locking call and the mutex-unlocking call, whether the mutex's scope is intra-process or inter-process. A thread calling to lock the mutex either gets exclusive access to the code starting from the successful locking until its call to unlock the mutex, or it waits until the mutex is unlocked by the thread that locked it.

Mutexes have ownership, unlike semaphores. Only the thread that locked a mutex, (that is, the owner of the mutex), should unlock it.

If a thread waiting for a mutex receives a signal, upon return from the signal handler, the thread resumes waiting for the mutex as if there was no interrupt.

**Caveats**

Mutexes are almost like data – they can be embedded in data structures, files, dynamic or static memory, and so forth. Hence, they are easy to introduce into a program. However, too many mutexes can degrade performance and scalability of the application. Because too few mutexes can hinder the concurrency of the application, they should be introduced with care. Also, incorrect usage (such as recursive calls, or violation of locking order, and so forth) can lead to deadlocks, or worse, data inconsistencies.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`mmap`(2), `shmop`(2), `mutex_destroy`(3THR), `mutex_init`(3THR), `mutex_lock`(3THR), `mutex_trylock`(3THR), `mutex_unlock`(3THR), `pthread_mutex_destroy`(3THR), `pthread_mutex_init`(3THR), `pthread_mutex_lock`(3THR), `pthread_mutex_trylock`(3THR), `pthread_mutex_unlock`(3THR), `pthread_create`(3THR), `pthread_mutexattr_init`(3THR), `attributes`(5), `standards`(5)

## NOTES

In the current implementation of threads, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `mutex_lock()` `mutex_unlock()`, `pthread_mutex_trylock()`, and `mutex_trylock()` do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return `EINVAL.` Interfaces for mutexes with an invalid type have unspecified behavior.

By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined.

`USYNC_THREAD` does not support multiple mapplings to the same logical synch object. If you need to `mmap()` a synch object to different locations within the same address space, then the synch object should be initialized as a shared object `USYNC_PROCESS` for Solaris, and `PTHREAD_PROCESS_PRIVATE` for POSIX.

**NAME**

mutex_init, mutex_destroy, mutex_lock, mutex_trylock, mutex_unlock – mutual exclusion locks

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...[ *library*... ]
#include <thread.h>
#include <synch.h>
int **mutex_init**(mutex_t **mp*, int *type*, void * *arg*);

int **mutex_lock**(mutex_t **mp*);

int **mutex_trylock**(mutex_t **mp*);

int **mutex_unlock**(mutex_t **mp*);

int **mutex_destroy**(mutex_t **mp*);

**DESCRIPTION**

Mutual exclusion locks (mutexes) prevent multiple threads from simultaneously executing critical sections of code which access shared data (that is, mutexes are used to serialize the execution of threads). All mutexes must be global. A successful call for a mutex lock by way of `mutex_lock()` will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks it by way of `mutex_unlock()` . Threads within the same process or within other processes can share mutexes.

Mutexes can synchronize threads within the same process or in other processes. Mutexes can be used to synchronize threads between processes if the mutexes are allocated in writable memory and shared among the cooperating processes (see `mmap`(2) ), and have been initialized for this task.

**Initialize**

Mutexes are either intra-process or inter-process, depending upon the argument passed implicitly or explicitly to the initialization of that mutex. A statically allocated mutex does not need to be explicitly initialized; by default, a statically allocated mutex is initialized with all zeros and its scope is set to be within the calling process.

For inter-process synchronization, a mutex needs to be allocated in memory shared between these processes. Since the memory for such a mutex must be allocated dynamically, the mutex needs to be explicitly initialized using `mutex_init()` .

The `mutex_init()` function initializes the mutex referenced by *mp* with the type specified by `type` . Upon successful initialization the state of the mutex becomes initialized and unlocked. No current type uses *arg* although a future type may specify additional behavior parameters by way of *arg* . `type` may be one of the following:

`USYNC_THREAD` The mutex can synchronize threads only in this process. *arg* is ignored.

`USYNC_PROCESS` The mutex can synchronize threads in this process and other processes. *arg* is ignored. The object initialized with this attribute must be allocated in memory shared between processes, either in System V shared memory (see `shmop`(2) ) or in memory mapped to a file (see `mmap`(2) ). If the object is not allocated in such shared memory, it will not be shared between processes.

`USYNC_PROCESS_ROBUST` The mutex can synchronize threads in this process and other processes robustly. At the time of process death, if the lock is held by the process, it is unlocked. The next owner of this mutex will acquire it with an error return of `EOWNERDEAD.` Note that the application must always check the return code from `mutex_lock( )` for a mutex of this type. The new owner of this mutex should then attempt to make the state protected by the mutex consistent, since this state could have been left inconsistent when the last owner died. If the new owner is able to make the state consistent, it should re-initialize the mutex and then unlock the mutex. If the new owner is not able to make the state consistent, for whatever reason, it should not re-initialize the mutex, but should just unlock the mutex. If the latter event occurs, all waiters will be woken up and all subsequent calls to `mutex_lock( )` will fail in acquiring the mutex with an error code of `ENOTRECOVERABLE.` mutex can be made consistent by un-initializing the mutex (`mutex_destroy( )` ) and re-initializing it (`mutex_init( )` ). If the process which got the lock with `EOWNERDEAD` died, the next owner will get the lock with an error return of `EOWNERDEAD.` *arg* is ignored. The object initialized with this attribute must be allocated in memory shared between processes, either in System V shared memory (see `shmop`(2) ) or in memory mapped to a file (see `mmap`(2) ) and memory must be zeroed before initialization. All the processes interested in the robust lock must call `mutex_init( )` at least once to register robust mutex with the system and potentially initialize it. If the object is not allocated in such shared memory, it will not be shared between processes. If `mutex_init()` is called on a previously initialized mutex `mutex_init()` will not re-initialize the mutex.

Initializing mutexes can also be accomplished by allocating in zeroed memory (default), in which case, a `type` of `USYNC_THREAD` is assumed. The same mutex must not be simultaneously initialized by multiple threads. A mutex lock must not be re-initialized while in use by other threads. If default mutex attributes are used, the macro `DEFAULTMUTEX` can be used to initialize mutexes that are statically allocated.

Default mutex initialization (intra-process):

```
mutex_t mp;
mutex_init(&mp, NULL, NULL);
```

*OR*

```
mutex_init(&mp, USYNC_THREAD, NULL);
```

*OR*

```
mutex_t  mp  =  DEFAULTMUTEX;
```

*OR*

```
mutex_t mp;
mp = calloc(1, sizeof (mutex_t));
```

*OR*

```
mutex_t mp;

mp = malloc(sizeof (mutex_t));

memset(mp, 0, sizeof (mutex_t));
```

Customized mutex initialization (inter-process):

```
mutex_init(&mp, USYNC_PROCESS, NULL);
```

Customized mutex initialization (inter-process):

```
mutex_init(&mp, USYNC_PROCESS_ROBUST, NULL);
```

**Lock and Unlock**

A critical section of code is enclosed by a the call to lock the mutex and the call to unlock the mutex to protect it from simultaneous access by multiple threads. Only one thread at a time may possess mutually exclusive access to the critical section of code that is enclosed by the mutex-locking call and the mutex-unlocking call, whether the mutex's scope is intra-process or inter-process. A thread calling to lock the mutex either gets exclusive access to the code starting from the successful locking until its call to unlock the mutex, or it waits until the mutex is unlocked by the thread that locked it.

Mutexes have ownership, unlike semaphores. Although any thread, within the scope of a mutex, can get an unlocked mutex and lock access to the same critical section of code, only the thread that locked a mutex should unlock it.

If a thread waiting for a mutex receives a signal, upon return from the signal handler, the thread resumes waiting for the mutex as if there was no interrupt. A mutex protects code, not data; therefore, strongly bind a mutex with the data by putting both within the same structure, or at least within the same procedure.

A call to `mutex_lock( )` locks the mutex object referenced by *mp* . If the mutex is already locked, the calling thread blocks until the mutex is freed; this will return with the mutex object referenced by *mp* in the locked state with the calling thread as its owner. If the current owner of a mutex tries to relock the mutex, it will result in deadlock.

`mutex_trylock( )` is the same as `mutex_lock( )` , respectively, except that if the mutex object referenced by *mp* is locked (by any thread, including the current thread), the call returns immediately with an error.

`mutex_unlock( )` are called by the owner of the mutex object referenced by *mp* to release it. The mutex must be locked and the calling thread must be the one that last locked the mutex (the owner). If there are threads blocked on the mutex object referenced by *mp* when `mutex_unlock( )` is called, the *mp* is freed, and the scheduling policy will determine which thread gets the mutex. If the calling thread is not the owner of the lock, no error status is returned, and the behavior of the program is undefined.

**Destroy**

`mutex_destroy( )` destroys the mutex object referenced by *mp* ; the mutex object becomes uninitialized. The space used by the destroyed mutex variable is not freed. It needs to be explicitly reclaimed.

## RETURN VALUES

If successful, these functions return `0` . Otherwise, an error number is returned.

## ERRORS

These functions may fail if:

`EFAULT` *mp* points to an illegal address.

The `mutex_init( )` function will fail if:

`EINVAL` The value specified by `type` is invalid.

The `mutex_init( )` function will fail for `USYNC_PROCESS_ROBUST` type mutex if:

`EBUSY` The mutex pointed to by *mp* was already initialized. An attempt to re-initialize a mutex previously initialized, but not yet destroyed.

The `mutex_trylock( )` function will fail if:

EBUSY The mutex pointed to by *mp* was already locked.

The mutex_lock( ) or mutex_trylock( ) functions will fail for USYNC_PROCESS_ROBUST type mutex if:

EOWNERDEAD
: The last owner of this mutex died while holding the mutex. This mutex is now owned by the caller. The caller must now attempt to make the state protected by the mutex consistent. If it is able to cleanup the state, then it should re-initialize the mutex (see mutex_init( ) ) and unlock the mutex. Subsequent calls to mutex_lock( ) will behave normally, as before. If the caller is not able to cleanup the state, the mutex should not be re-initialized, it should be unlocked. Subsequent calls to mutex_lock( ) will fail to acquire the mutex, with the error code, ENOTRECOVERABLE. If the owner who got the lock with EOWNERDEAD died, the next owner will get the lock with EOWNERDEAD.

ELOCKUNMAPPED
: The last owner of this mutex unmaped the mutex while holding the mutex. This mutex is now owned by the caller. The caller must now attempt to make the state protected by the mutex consistent. If it is able to cleanup the state, then it should re-initialize the mutex unlock the mutex. See mutex_init(3THR) . Subsequent calls to mutex_lock( ) will behave normally, as before. If the caller is not able to cleanup the state, the mutex should not be re-initialized. Subsequent calls to mutex_lock( ) will fail to acquire the mutex with the error code, ENOTRECOVERABLE.

ENOTRECOVERABLE
: The mutex trying to be acquired is protecting state which has been left irrecoverable by the mutex's last owner, which died while holding the lock. The mutex has not been acquired. This condition can occur when the lock was previously acquired with EOWNERDEAD or ELOCKUNMAPPED and the owner was not able to cleanup the state and unlocked the mutex with out making the mutex consistent.

**EXAMPLES**

**Single Gate**

The following example uses one global mutex as a gate-keeper to permit each thread exclusive sequential access to the code within the user-defined function "change_global_data." This type of synchronization will protect the state of shared data, but it also prohibits parallelism.

```
/* cc thisfile.c -lthread */
#define _REENTRANT
#include <stdio.h>
#include <thread.h>
#define NUM_THREADS 12
void *change_global_data(void *);     /*  for thr_create()   */
main(int argc,char * argv[]) {
       int i=0;
       for (i=0; i< NUM_THREADS; i++) {
               thr_create(NULL, 0, change_global_data, NULL, 0, NULL);
       }
       while ((thr_join(NULL, NULL, NULL) == 0));
}

void * change_global_data(void *null) {
       static mutex_t Global_mutex;
       static int Global_data = 0;
       mutex_lock(&Global_mutex);
       Global_data++;
       sleep(1);
       printf("%d is global data\
",Global_data);
       mutex_unlock(&Global_mutex);
       return NULL;
}
```

**Multiple Instruction Single Data**

The previous example, the mutex, the code it owns, and the data it protects was enclosed in one function. The next example uses C++ features to accommodate many functions that use just one mutex to protect one data:

```
/* CC thisfile.c -lthread   use C++ to compile*/

#define _REENTRANT
#include <stdlib.h>
#include <stdio.h>
#include <thread.h>
#include <errno.h>
#include <iostream.h>
#define NUM_THREADS 16
void *change_global_data(void *);     /*  for thr_create()   */

class Mutected {
       private:
               static mutex_t       Global_mutex;
               static int          Global_data;
       public:
               static int          add_to_global_data(void);
               static int          subtract_from_global_data(void);
```

```
};

int Mutected::Global_data = 0;
mutex_t Mutected::Global_mutex;

int Mutected::add_to_global_data()  {
       mutex_lock(&Global_mutex);
       Global_data++;
       mutex_unlock(&Global_mutex);
       return Global_data;
}

int Mutected::subtract_from_global_data()   {
       mutex_lock(&Global_mutex);
       Global_data--;
       mutex_unlock(&Global_mutex);
       return Global_data;
}

void
main(int argc,char * argv[])  {
       int i=0;
       for (i=0;i< NUM_THREADS;i++)  {
            thr_create(NULL,0,change_global_data,NULL,0,NULL);
       }
       while ((thr_join(NULL,NULL,NULL) == 0));
}

void * change_global_data(void *) {
       static int switcher = 0;
       if ((switcher++ % 3) == 0)   /* one-in-three threads subtracts */
               cout << Mutected::subtract_from_global_data() << endl;
       else
               cout << Mutected::add_to_global_data() << endl;
       return NULL;
}
```

**Interprocess Locking**

A mutex can protect data that is shared among processes. The mutex would need to be initialized as USYNC_PROCESS. One process initializes the process-shared mutex and writes it to a file to be mapped into memory by all cooperating processes (see mmap(2) ). Afterwards, other independent processes can run the same program (whether concurrently or not) and share mutex-protected data.

```
/* cc thisfile.c -lthread */
/* To execute, run the command line "a.out 0 & a.out 1" */

#define _REENTRANT
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <thread.h>
#define INTERPROCESS_FILE "ipc-sharedfile"
```

```
#define NUM_ADDTHREADS 12
#define NUM_SUBTRACTTHREADS 10
#define INCREMENT '0'
#define DECREMENT '1'
typedef struct {
               mutex_t     Interprocess_mutex;
               int         Interprocess_data;
} buffer_t;
buffer_t *buffer;


void *add_interprocess_data(), *subtract_interprocess_data();
void create_shared_memory(), test_argv();
int zeroed[sizeof(buffer_t)];
int ipc_fd, i=0;


void
main(int argc,char * argv[]){
       test_argv(argv[1]);

       switch (*argv[1])  {
       case INCREMENT:
             create_shared_memory();
             ipc_fd = open(INTERPROCESS_FILE, O_RDWR);
             buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
                    PROT_READ|PROT_WRITE, MAP_SHARED, ipc_fd, 0);
             buffer->Interprocess_data = 0;
             mutex_init(&buffer->Interprocess_mutex, USYNC_PROCESS,0);
             for (i=0; i< NUM_ADDTHREADS; i++)
             thr_create(NULL, 0, add_interprocess_data, argv[1],
                     0, NULL);
             break;

       case DECREMENT:
             while((ipc_fd = open(INTERPROCESS_FILE, O_RDWR)) == -1)
                     sleep(1);
             buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
                     PROT_READ|PROT_WRITE, MAP_SHARED, ipc_fd, 0);
             for (i=0; i< NUM_SUBTRACTTHREADS; i++)
             thr_create(NULL, 0, subtract_interprocess_data, argv[1],
                     0, NULL);
             break;
      } /* end switch */

      while ((thr_join(NULL,NULL,NULL) == 0));
} /* end main */

void *add_interprocess_data(char argv_1[]){
      mutex_lock(&buffer->Interprocess_mutex);
      buffer->Interprocess_data++;
      sleep(2);
      printf("%d is add-interprocess data, and %c is argv1\
",
              buffer->Interprocess_data, argv_1[0]);
```

```
       mutex_unlock(&buffer->Interprocess_mutex);
       return NULL;
 }

 void *subtract_interprocess_data(char argv_1[]) {
       mutex_lock(&buffer->Interprocess_mutex);
       buffer->Interprocess_data--;
       sleep(2);
       printf("%d is subtract-interprocess data, and %c is argv1\
",
               buffer->Interprocess_data, argv_1[0]);
       mutex_unlock(&buffer->Interprocess_mutex);
       return NULL;
 }

 void create_shared_memory(){
       int i;
       ipc_fd = creat(INTERPROCESS_FILE, O_CREAT|O_RDWR );
       for (i=0; i<sizeof(buffer_t); i++){
               zeroed[i] = 0;
               write(ipc_fd, &zeroed[i],2);
       }
       close(ipc_fd);
       chmod(INTERPROCESS_FILE, S_IRWXU|S_IRWXG|S_IRWXO);
 }

 void test_argv(char argv1[])   {
       if (argv1 == NULL)   {
       printf("use 0 as arg1 for initial process\
  \\
       or use 1 as arg1 for the second process\
");
       exit(NULL);
       }
 }
```

In this example, run the command line

`a.out 0 & a.out 1`

**Solaris Interprocess Robust Locking**

A mutex can protect data that is shared among processes robustly. The mutex would need to be initialized as `USYNC_PROCESS_ROBUST`. One process initializes the robust process-shared mutex and writes it to a file to be mapped into memory by all cooperating processes (see `mmap`(2) ). Afterwards, other independent processes can run the same program (whether concurrently or not) and share mutex-protected data.

The following example shows how to use a `USYNC_PROCESS_ROBUST` type mutex.

```
/* cc thisfile.c -lthread */
/* To execute, run the command line "a.out & a.out 1" */
```

```
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <thread.h>
#define INTERPROCESS_FILE "ipc-sharedfile"
typedef struct {
          mutex_t   Interprocess_mutex;
          int       Interprocess_data;
} buffer_t;
buffer_t *buffer;
int make_date_consistent();
void create_shared_memory();
int zeroed[sizeof(buffer_t)];
int ipc_fd, i=0;
main(int argc,char * argv[])  {
          int rc;
          if (argc > 1) {
              while((ipc_fd = open(INTERPROCESS_FILE, O_RDWR)) == -1)
                  sleep(1);
              buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
                        PROT_READ|PROT_WRITE, MAP_SHARED, ipc_fd, 0);
              mutex_init(&buffer->Interprocess_mutex,
                          USYNC_PROCESS_ROBUST,0);
          } else {
              create_shared_memory();
              ipc_fd = open(INTERPROCESS_FILE, O_RDWR);
              buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
                    PROT_READ|PROT_WRITE, MAP_SHARED, ipc_fd, 0);
              buffer->Interprocess_data = 0;
              mutex_init(&buffer->Interprocess_mutex,
                          USYNC_PROCESS_ROBUST,0);
          }
          for(;;) {
              rc = mutex_lock(&buffer->Interprocess_mutex);
              switch (rc) {
                  case EOWNERDEAD:
                    /* lock acquired.
                     * last owner died holding the lock, try to make
                     * the state associated with the mutex consistent.
                     * If so, make the robust lock consistent by
                     * re-initializing it.
                     */
                    if (make_data_consistent())
                        mutex_init(&buffer->Interprocess_mutex,
                               USYNC_PROCESS_ROBUST,0);
                    mutex_unlock(&buffer->Interprocess_mutex);
                    case ENOTRECOVERABLE:
                      /* lock not acquired.
                       * last owner got the mutex with EOWNERDEAD
                       * mutex is not consistent (and data?),
                       * so return from here
                       */
                      exit(1);
                      break;
```

```
                            case 0:
                                /* no error - data is consistent */
                                /* do something with data */
                                mutex_unlock(&buffer->Interprocess_mutex);
                                break;
                        }
                    }
          } /* end main */
          void create_shared_memory() {
                int i;
                ipc_fd = creat(INTERPROCESS_FILE, O_CREAT|O_RDWR );
                for (i=0; i<sizeof(buffer_t); i++) {
                     zeroed[i] = 0;
                     write(ipc_fd, &zeroed[i],2);
                }
                close(ipc_fd);
                chmod(INTERPROCESS_FILE, S_IRWXU|S_IRWXG|S_IRWXO);
           }

           /* return 1 if able to make data consistent, otherwise 0. */
           int make_data_consistent () {
                 buffer->Interprocess_data = 0;
                 return (1);
           }
```

**Dynamically Allocated Mutexes**

The following example allocates and frees memory in which a mutex is embedded.

```
struct record {
            int field1;
            int field2;
            mutex_t m;
} *r;
r = malloc(sizeof(struct record));
mutex_init(&r->m, USYNC_THREAD, NULL);
/*
  * The fields in this record are accessed concurrently
  * by acquiring the embedded lock.
  */
```

The thread execution in this example is as follows:

| *Thread 1 executes:* | *Thread 2 executes:* |
| --- | --- |
| `...` | `...` |
| `mutex_lock(&r->m);` | `mutex_lock(&r->m);` |
| `r->field1++;` | `localvar = r->field1;` |
| `mutex_unlock(&r->m);` | `mutex_unlock(&r->m);` |
| `...` | `...` |

Later, when a thread decides to free the memory pointed to by *r* , the thread should call `mutex_destroy` () on the mutexes in this memory.

In the following example, the main thread can do a `thr_join` () on both of the above threads. If there are no other threads using the memory in *r* , the main thread can now safely free *r* :

```
for (i = 0; i < 2; i++)
       thr_join(0, 0, 0);
mutex_destroy(&r->m);     /* first destroy mutex */
free(r);                 /* Then free memory */
```

If the mutex is not destroyed, the program could have memory leaks.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`mmap`(2) , `shmop`(2) , `mutex`(3THR) , `attributes`(5) , `standards`(5)

**NOTES**

Currently, the only supported policy is `SCHED_OTHER.` In Solaris, under the `SCHED_OTHER` policy, there is no established order in which threads are unblocked.

In the current implementation of threads, `mutex_lock()` , `mutex_unlock()` , and `mutex_trylock()` do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return `EINVAL.` Interfaces for mutexes with an invalid type have unspecified behavior.

Uninitialized mutexes which are allocated locally may contain junk data. Such mutexes need to be initialized using `mutex_init()` .

By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined.

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`sleep`(3C), `attributes`(5), `time`(3HEAD)

## 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*);

**IA**

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 will 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 `libthread_db`(3THR). 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. <proc_service.h> lists the C declarations of these routines

## FUNCTIONS

| Name | Description |
|---|---|
| `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. |

**IA**

| | |
|---|---|
| `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

`libthread_db`(3THR), `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 (LWP s) 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**

libthread(3THR) , libthread_db(3THR) , proc_service(3PROC) , libthread_db(3LIB) , attributes(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) , `libthread`(3THR) , `libthread_db`(3THR) , `proc_service`(3PROC) , `libthread_db`(3LIB) , `attributes`(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**

`libthread`(3THR) , `libthread_db`(3THR) , `proc_service`(3PROC) , `libthread_db`(3LIB) , `attributes`(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

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

`libthread_db( )` 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.

`ps_lstop( )` and `ps_lcontinue( )` stop and resume a single lightweight process (LWP ) within the target process *ph* . They are not currently used by `libthread_db( )` .

`ps_lrolltoaddr( )` 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( )` . `ps_lrolltoaddr( )` 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* .

`ps_kill( )` directs the signal `signum` to the target process for which the handle is *ph* . `ps_kill( )` 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) , `libthread`(3THR) , `libthread_db`(3THR) , `proc_service`(3PROC) , `libthread_db`(3LIB) , `attributes`(5)

**NAME**

pthread_atfork – register fork handlers

**SYNOPSIS**

cc –mt [ *flag...* ] *file...*– lpthread [ *library...* ]
#include <sys/types.h>
#include <unistd.h>
int **pthread_atfork**(void (**prepare*) (void), void (**parent*) (void), void (**child*) (void));

**DESCRIPTION**

The `pthread_atfork()` function declares fork handlers to be called prior to and following `fork`(2), within the thread that called `fork()`. The order of calls to `pthread_atfork()` is significant.

Before `fork()` processing begins, the *prepare* fork handler is called. The *prepare* handler is not called if its address is `NULL`.

The *parent* fork handler is called after `fork()` processing finishes in the parent process, and the *child* fork handler is called after `fork()` processing finishes in the child process. If the address of *parent* or *child* is `NULL`, then its handler is not called.

The *prepare* fork handler is called in LIFO (last-in first-out) order, whereas the *parent* and *child* fork handlers are called in FIFO (first-in first-out) order. This calling order allows applications to preserve locking order.

**RETURN VALUES**

Upon successful completion, `pthread_atfork()` returns `0`. Otherwise, an error number is returned.

**ERRORS**

The `pthread_atfork()` function will fail if:

`ENOMEM` Insufficient table space exists to record the fork handler addresses.

**USAGE**

Solaris threads do not offer `pthread_atfork()` functionality, though a Solaris threads application may call this interface, since the two thread APIs are interoperable. See `fork`(2).

**EXAMPLES**

**EXAMPLE 1** make a library safe with respect to `fork()`

All multithreaded applications that call `fork()` in a POSIX threads program and do more than simply call `exec`(2) in the child of the fork need to ensure that the child is protected from deadlock.

Since the "fork-one" model results in duplicating only the thread that called `fork()`, it is possible that at the time of the call another thread in the parent owns a lock. This thread is not duplicated in the child, so no thread will unlock this lock in the child. Deadlock occurs if the single thread in the child needs this lock.

The problem is more serious with locks in libraries. Since a library writer does not know if the application using the library calls `fork()`, the library must protect itself from such a deadlock scenario. If the application that links with this library calls fork( ) and does not call exec( ) in the child, and if it needs a library lock that may be held by some other thread in the parent that is inside the library at the time of the fork, the application deadlocks inside the library.

The following describes how to make a library safe with respect to fork( ) by using pthread_atfork( ).

1. Identify all locks used by the library (for example {L1, . . .Ln}). Identify also the locking order for these locks (for example {L1 . . .Ln}, as well.)
2. Add a call to pthread_atfork(f1, f2, f3) in the library's *.init* section. f1, f2, f3 are defined as follows:

```
f1( )
 {
         /* ordered in lock order */
         pthread_mutex_lock(L1);
         pthread_mutex_lock( . . .);
         pthread_mutex_lock(Ln);
 }

 f2( )
 {
         pthread_mutex_unlock(L1);
         pthread_mutex_unlock( . . .);
         pthread_mutex_unlock(Ln);
 }

 f3( )
 {
         pthread_mutex_unlock(L1);
         pthread_mutex_unlock( . . .);
         pthread_mutex_unlock(Ln);
 }
```

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

exec(2), fork(2), atexit(3C), attributes(5), standards(5)

**NAME**

pthread_attr_getdetachstate, pthread_attr_setdetachstate – get or set detachstate attribute

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
```

**DESCRIPTION**

The *detachstate* attribute controls whether the thread is created in a detached state. If the thread is created detached, then use of the ID of the newly created thread by the `pthread_detach()` or `pthread_join()` function is an error.

The `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()`, respectively, set and get the *detachstate* attribute in the *attr* object.

The *detachstate* can be set to either `PTHREAD_CREATE_DETACHED` or `PTHREAD_CREATE_JOINABLE`. A value of `PTHREAD_CREATE_DETACHED` causes all threads created with *attr* to be in the detached state, whereas using a value of `PTHREAD_CREATE_JOINABLE` causes all threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute is `PTHREAD_CREATE_JOINABLE`.

**RETURN VALUES**

Upon successful completion, `pthread_attr_setdetachstate()` and `pthread_attr_getdetachstate()` return a value of `0`. Otherwise, an error number is returned to indicate the error.

The `pthread_attr_getdetachstate()` function stores the value of the *detachstate* attribute in *detachstate* if successful.

**ERRORS**

The `pthread_attr_setdetachstate()` or `pthread_attr_getdetachstate()` functions may fail if:

`EINVAL` *attr* or *detachstate* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_attr_init`(3THR) , `pthread_attr_setstackaddr`(3THR) , `pthread_attr_setstacksize`(3THR) , `pthread_create`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

pthread_attr_getguardsize, pthread_attr_setguardsize – get or set the thread guardsize attribute

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]
#include <pthread.h>
int **pthread_attr_getguardsize**(const pthread_attr_t **attr*, size_t **guardsize*);

int **pthread_attr_setguardsize**(pthread_attr_t **attr*, size_t *guardsize*);

**DESCRIPTION**

The *guardsize* attribute controls the size of the guard area for the created thread's stack. The *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results (possibly in a `SIGSEGV` signal being delivered to the thread).

The *guardsize* attribute is provided to the application for two reasons:

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads will never overflow their stack, can save system resources by turning off guard areas.
2. When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

The `pthread_attr_getguardsize( )` function gets the *guardsize* attribute in the *attr* object. This attribute is returned in the *guardsize* parameter.

The `pthread_attr_setguardsize( )` function sets the *guardsize* attribute in the *attr* object. The new value of this attribute is obtained from the *guardsize* parameter. If *guardsize* is `0` , a guard area will not be provided for threads created with *attr* . If *guardsize* is greater than `0` , a guard area of at least size *guardsize* bytes is provided for each thread created with *attr* .

A conforming implementation is permitted to round up the value contained in *guardsize* to a multiple of the configurable system variable `PAGESIZE.` If an implementation rounds up the value of *guardsize* to a multiple of `PAGESIZE,` a call to `pthread_attr_getguardsize( )` specifying *attr* will store in the *guardsize* parameter the guard size specified by the previous `pthread_attr_setguardsize( )` function call.

The default value of the *guardsize* attribute is `PAGESIZE` bytes. The actual value of `PAGESIZE` is implementation-dependent and may not be the same on all implementations.

If the *stackaddr* attribute has been set (that is, the caller is allocating and managing its own thread stacks), the *guardsize* attribute is ignored and no protection will be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

**RETURN VALUES**

If successful, the `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions return `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_attr_getguardsize()` and `pthread_attr_setguardsize()` functions will fail if:

`EINVAL` The attribute *attr* is invalid.

`EINVAL` The parameter *guardsize* is invalid.

`EINVAL` The parameter *guardsize* contains an invalid value.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO**

`sysconf`(3C) , `pthread_attr_init`(3THR) , `attributes`(5)

**NAME**

pthread_attr_getinheritsched, pthread_attr_setinheritsched – get or set inheritsched attribute

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*- lpthread [ *library...* ]

```
#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);

int pthread_attr_getinheritsched(const pthread_attr_t *attr, int *inheritsched);
```

**DESCRIPTION**

The functions `pthread_attr_setinheritsched()` and `pthread_attr_getinheritsched()` , respectively, set and get the *inheritsched* attribute in the *attr* argument.

When the attribute objects are used by `pthread_create()` , the *inheritsched* attribute determines how the other scheduling attributes of the created thread are to be set:

`PTHREAD_INHERIT_SCHED` Specifies that the scheduling policy and associated attributes are to be inherited from the creating thread, and the scheduling attributes in this *attr* argument are to be ignored.

`PTHREAD_EXPLICIT_SCHED` Specifies that the scheduling policy and associated attributes are to be set to the corresponding values from this attribute object.

The symbols `PTHREAD_INHERIT_SCHED` and `PTHREAD_EXPLICIT_SCHED` are defined in the header `<pthread.h>` .

**RETURN VALUES**

If successful, the `pthread_attr_setinheritsched()` and `pthread_attr_getinheritsched()` functions return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_attr_setinheritsched()` or `pthread_attr_getinheritsched()` functions may fail if:

`EINVAL` *attr* or *inheritsched* is invalid.

**USAGE**

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()` . Using these routines does not affect the current running thread.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_attr_init(3THR) , pthread_attr_setscope(3THR) , pthread_attr_setschedpolicy(3THR) , pthread_attr_setschedparam(3THR) , pthread_create(3THR) , pthread_setsched_param(3THR) , attributes(5) , standards(5)

**NAME**

pthread_attr_getschedparam, pthread_attr_setschedparam – get or set schedparam attribute

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

#include <pthread.h>
int **pthread_attr_setschedparam**(pthread_attr_t **attr*, const struct sched_param **param*);

int **pthread_attr_getschedparam**(const pthread_attr_t **attr*, struct sched_param **param*);

**DESCRIPTION**

The functions `pthread_attr_setschedparam()` and `pthread_attr_getschedparam()`, respectively, set and get the scheduling parameter attributes in the *attr* argument. The contents of the *param* structure are defined in `<sched.h>`. For the `SCHED_FIFO` and `SCHED_RR` policies, the only required member of *param* is *sched_priority* .

**RETURN VALUES**

If successful, the `pthread_attr_setschedparam()` and `pthread_attr_getschedparam()` functions return `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_attr_setschedparam()` function may fail if:

`EINVAL` *attr* is invalid.

The `pthread_attr_getschedparam()` function may fail if:

`EINVAL` *attr* or *param* is invalid.

**USAGE**

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()`. Using these routines does not affect the current running thread.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_attr_init`(3THR) , `pthread_attr_setscope`(3THR) , `pthread_attr_setinheritsched`(3THR) , `pthread_attr_setschedpolicy`(3THR) , `pthread_create`(3THR) , `pthread_setschedparam`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

pthread_attr_getschedpolicy, pthread_attr_setschedpolicy – get or set schedpolicy attribute

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*- lpthread [ *library...* ]

```
#include <pthread.h>
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
```

**DESCRIPTION**

The functions `pthread_attr_setschedpolicy()` and `pthread_attr_getschedpolicy()` , respectively, set and get the *schedpolicy* attribute in the *attr* argument.

The supported values of *policy* include `SCHED_FIFO,` `SCHED_RR` and `SCHED_OTHER,` which are defined by the header `<sched.h>` . When threads executing with the scheduling policy `SCHED_FIFO` or `SCHED_RR` are waiting on a mutex, they acquire the mutex in priority order when the mutex is unlocked.

**RETURN VALUES**

If successful, the `pthread_attr_setschedpolicy()` and `pthread_attr_getschedpolicy()` functions return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_attr_setschedpolicy()` or `pthread_attr_getschedpolicy()` function may fail if:

`EINVAL` *attr* or *policy* is invalid.

**USAGE**

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create()` . Using these routines does not affect the current running thread.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_attr_init`(3THR) , `pthread_attr_setscope`(3THR) , `pthread_attr_setinheritsched`(3THR) , `pthread_attr_setschedparam`(3THR) , `pthread_create`(3THR) , `pthread_setschedparam`(3THR) , `attributes`(5) , `standards`(5)

## NAME

pthread_attr_getscope, pthread_attr_setscope – get or set contentionscope attribute

## SYNOPSIS

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

```
#include <pthread.h>
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);

int pthread_attr_getscope(const pthread_attr_t *attr, int *contentionscope);
```

## DESCRIPTION

The `pthread_attr_setscope( )` and `pthread_attr_getscope( )` functions are used to set and get the *contentionscope* attribute in the *attr* object.

The `pthread_attr_setscope( )` and `pthread_attr_getscope( )` functions set and get the *contentionscope* thread attribute in the *attr* object. The *contentionscope* value may be set to the following:

`PTHREAD_SCOPE_SYSTEM` Indicates system scheduling contention scope. This thread is permanently "bound" to an LWP, and is also called a bound thread.

`PTHREAD_SCOPE_PROCESS` Indicates process scheduling contention scope. This thread is not "bound" to an LWP, and is also called an unbound thread. `PTHREAD_SCOPE_PROCESS` , or unbound, is the default.

`PTHREAD_SCOPE_SYSTEM` and `PTHREAD_SCOPE_PROCESS` are defined by the header `<pthread.h>` .

## RETURN VALUES

If successful, the `pthread_attr_setscope( )` and `pthread_attr_getscope( )` functions return `0` . Otherwise, an error number is returned to indicate the error.

## ERRORS

The `pthread_attr_setscope( )` , or `pthread_attr_getscope( )` , function may fail if:

`EINVAL` *attr* or *contentionscope* is invalid.

## USAGE

After these attributes have been set, a thread can be created with the specified attributes using `pthread_create( )` . Using these routines does not affect the current running thread.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_attr_init(3THR) , pthread_attr_setinheritsched(3THR) , pthread_attr_setschedpolicy(3THR) , pthread_attr_setschedparam(3THR) , pthread_create(3THR) , pthread_setschedparam(3THR) , attributes(5) , standards(5)

**NAME**

pthread_attr_getstackaddr, pthread_attr_setstackaddr – get or set stackaddr attribute

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

#include <pthread.h>
int **pthread_attr_setstackaddr**(pthread_attr_t **attr*, void **stackaddr*);

int **pthread_attr_getstackaddr**(const pthread_attr_t **attr*, void ***stackaddr*);

**DESCRIPTION**

The functions `pthread_attr_setstackaddr()` and `pthread_attr_getstackaddr()` , respectively, set and get the thread creation *stackaddr* attribute in the *attr* object. The *stackaddr* default is `NULL` . See `pthread_create`(3THR) .

The *stackaddr* attribute specifies the location of storage to be used for the created thread's stack. The size of the storage is at least `PTHREAD_STACK_MIN`.

**RETURN VALUES**

Upon successful completion, `pthread_attr_setstackaddr()` and `pthread_attr_getstackaddr()` return a value of `0` . Otherwise, an error number is returned to indicate the error.

If successful, the `pthread_attr_getstackaddr()` function stores the *stackaddr* attribute value in *stackaddr* .

**ERRORS**

The `pthread_attr_setstackaddr()` function may fail if:

`EINVAL` *attr* is invalid.

The `pthread_attr_getstackaddr()` function may fail if:

`EINVAL` *attr* or *stackaddr* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_attr_init`(3THR) , `pthread_attr_setdetachstate`(3THR) , `pthread_attr_setstacksize`(3THR) , `pthread_create`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

pthread_attr_getstacksize, pthread_attr_setstacksize – get or set stacksize attribute

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

#include <pthread.h>
int **pthread_attr_setstacksize**(pthread_attr_t **attr*, size_t *stacksize*);

int **pthread_attr_getstacksize**(const pthread_attr_t **attr*, size_t **stacksize*);

**DESCRIPTION**

The functions `pthread_attr_setstacksize()` and `pthread_attr_getstacksize()`, respectively, set and get the thread creation *stacksize* attribute in the *attr* object.

The *stacksize* attribute defines the minimum stack size (in bytes) allocated for the created threads stack. When the *stacksize* argument is `NULL`, the default stack size becomes `1` megabyte for 32-bit processes and `2` megabytes for 64-bit processes.

**RETURN VALUES**

Upon successful completion, `pthread_attr_setstacksize()` and `pthread_attr_getstacksize()` return a value of `0`. Otherwise, an error number is returned to indicate the error. The `pthread_attr_getstacksize()` function stores the *stacksize* attribute value in *stacksize* if successful.

**ERRORS**

The `pthread_attr_setstacksize()` or `pthread_attr_getstacksize()` function may fail if:

`EINVAL` *attr* or *stacksize* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_attr_init`(3THR) , `pthread_attr_setstackaddr`(3THR) , `pthread_attr_setdetachstate`(3THR) , `pthread_create`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

pthread_attr_init, pthread_attr_destroy – initialize or destroy threads attribute object

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

#include <pthread.h>
int **pthread_attr_init**(pthread_attr_t **attr*);

int **pthread_attr_destroy**(pthread_attr_t **attr*);

**DESCRIPTION**

The function `pthread_attr_init( )` initializes a thread attributes object *attr* with the default value for all of the individual attributes used by a given implementation.

The resulting attribute object (possibly modified by setting individual attribute values), when used by `pthread_create( )` , defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to `pthread_create( )` .

The `pthread_attr_init( )` function initializes a thread attributes object ( *attr* ) with the default value for each attribute as follows:

| Attribute | Default Value | Meaning of Default |
|---|---|---|
| *contentionscope* | PTHREAD_SCOPE_PROCESS | resource competition within process |
| *detachstate* | PTHREAD_CREATE_JOINABLE | joinable by other threads |
| *stackaddr* | NULL | stack allocated by system |
| *stacksize* | NULL | 1 or 2 megabyte |
| *priority* | 0 | priority of the thread |
| *policy* | SCHED_OTHER | determined by system |
| *inheritsched* | PTHREAD_EXPLICIT_SCHED | scheduling policy and parameters not inherited but explicitly defined by the attribute object |
| *guardsize* | PAGESIZE | size of guard area for a thread's created stack |

The `pthread_attr_destroy( )` function destroys a thread attributes object ( *attr* ), which cannot be reused until it is reinitialized. An implementation may cause `pthread_attr_destroy( )` to set *attr* to an implementation-dependent invalid value. The behavior of using the attribute after it has been destroyed is undefined.

**RETURN VALUES**

Upon successful completion, `pthread_attr_init()` and `pthread_attr_destroy()` return a value of `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_attr_init()` function will fail if:

`ENOMEM` Insufficient memory exists to initialize the thread attributes object.

The `pthread_attr_destroy()` function may fail if:

`EINVAL` *attr* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`sysconf`(3C) , `pthread_attr_getdetachstate`(3THR) , `pthread_attr_getguardsize`(3THR) , `pthread_attr_getinheritsched`(3THR) , `pthread_attr_getschedparam`(3THR) , `pthread_attr_getschedpolicy`(3THR) , `pthread_attr_getscope`(3THR) , `pthread_attr_getstackaddr`(3THR) , `pthread_attr_getstacksize`(3THR) , `pthread_attr_setdetachstate`(3THR) , `pthread_attr_setguardsize`(3THR) , `pthread_attr_setinheritsched`(3THR) , `pthread_attr_setschedparam`(3THR) , `pthread_attr_setschedpolicy`(3THR) , `pthread_attr_setscope`(3THR) , `pthread_attr_setstackaddr`(3THR) , `pthread_attr_setstacksize`(3THR) , `pthread_create`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

pthread_cancel – cancel execution of a thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
int **pthread_cancel**(pthread_t *target_thread*);

**DESCRIPTION**

The `pthread_cancel()` function requests that *target_thread* be canceled.

By default, cancellation is deferred until *target_thread* reaches a cancellation point. See `cancellation`(3THR).

Cancellation cleanup handlers for *target_thread* are called when the cancellation is acted on. Upon return of the last cancellation cleanup handler, the thread-specific data destructor functions are called for *target_thread*. *target_thread* is terminated when the last destructor function returns.

The cancellation processing in *target_thread* runs asynchronously with respect to the calling thread returning from `pthread_cancel()`.

**RETURN VALUES**

If successful, the `pthread_cancel()` function returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_cancel()` function may fail if:

`ESRCH` No thread was found with an ID corresponding to that specified by the given thread ID, *target_thread*.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`cancellation`(3THR), `condition`(3THR), `pthread_cleanup_pop`(3THR), `pthread_cleanup_push`(3THR), `pthread_cond_wait`(3THR), `pthread_cond_timedwait`(3THR), `pthread_exit`(3THR), `pthread_join`(3THR), `pthread_setcancelstate`(3THR), `pthread_setcanceltype`(3THR), `pthread_testcancel`(3THR), `setjmp`(3C), `attributes`(5)

**NOTES**

See `cancellation`(3THR) for a discussion of cancellation concepts.

**NAME**

pthread_cleanup_pop – pop a thread cancellation cleanup handler

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
void **pthread_cleanup_pop**(int*execute*);

**DESCRIPTION**

`pthread_cleanup_pop( )` removes the cleanup handler routine at the top of the cancellation cleanup stack of the calling thread and executes it if *execute* is non-zero.

When the thread calls `pthread_cleanup_pop( )` with a non-zero *execute* argument, the argument at the top of the stack is popped and executed. An argument of `0` pops the handler without executing it.

The Solaris system generates a compile time error if `pthread_cleanup_push( )` does not have a matching `pthread-cleanup_pop( ).`

Be aware that using `longjmp( )` or `siglongjmp( )` to jump into or out of a push / pop pair can lead to trouble, as either the matching push or the matching pop statement might not get executed.

**RETURN VALUES**

The `pthread_cleanup_pop( )` function returns no value.

**ERRORS**

No errors are defined.

The `pthread_cleanup_pop( )` function will not return an error code of `EINTR`.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`cancellation`(3THR), `condition`(3THR), `pthread_cancel`(3THR), `pthread_cleanup_push`(3THR), `pthread_exit`(3THR), `pthread_join`(3THR), `pthread_setcancelstate`(3THR), `pthread_setcanceltype`(3THR), `pthread_testcancel`(3THR), `setjmp`(3C), `attributes`(5)

**NOTES**

See `cancellation`(3THR) for a discussion of cancellation concepts.

**NAME**

pthread_cleanup_push – push a thread cancellation cleanup handler

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
void **pthread_cleanup_push**(void (**handler*, void *),void **arg*);

**DESCRIPTION**

`pthread_cleanup_push( )` pushes the specified cancellation cleanup handler routine, *handler*, onto the cancellation cleanup stack of the calling thread.

When a thread exits or is canceled and its cancellation cleanup stack is not empty, the cleanup handlers are invoked with the argument *arg* in last in, first out (LIFO) order from the cancellation cleanup stack.

The Solaris system generates a compile time error if `pthread_cleanup_push( )` does not have a matching `pthread_cleanup_pop( )`.

Be aware that using `longjmp( )` or `siglongjmp( )` to jump into or out of a push / pop pair can lead to trouble, as either the matching push or the matching pop statement might not get executed.

**RETURN VALUES**

The `pthread_cleanup_push( )` function returns no value.

**ERRORS**

No errors are defined.

The `pthread_cleanup_push( )` function will not return an error code of `EINTR`.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO**

`cancellation`(3THR), `condition`(3THR), `longjmp`(3C), `pthread_cancel`(3THR), `pthread_cleanup_pop`(3THR), `pthread_exit`(3THR), `pthread_join`(3THR), `pthread_setcancelstate`(3THR), `pthread_setcanceltype`(3THR), `pthread_testcancel`(3THR), `attributes`(5)

**NOTES**

See `cancellation`(3THR) for a discussion of cancellation concepts.

**NAME**

pthread_condattr_getpshared, pthread_condattr_setpshared – get or set the process-shared condition variable attributes

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

```
#include <pthread.h>
int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *pshared);

int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
```

**DESCRIPTION**

The `pthread_condattr_getpshared()` function obtains the value of the *process-shared* attribute from the attributes object referenced by *attr* . The `pthread_condattr_setpshared()` function is used to set the *process-shared* attribute in an initialized attributes object referenced by *attr* .

The *process-shared* attribute is set to `PTHREAD_PROCESS_SHARED` to permit a condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated, even if the condition variable is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE`, the condition variable will only be operated upon by threads created within the same process as the thread that initialized the condition variable; if threads of differing processes attempt to operate on such a condition variable, the behavior is undefined. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-dependent.

**RETURN VALUES**

If successful, the `pthread_condattr_setpshared()` function returns `0` . Otherwise, an error number is returned to indicate the error.

If successful, the `pthread_condattr_getpshared()` function returns `0` and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_condattr_getpshared()` and `pthread_condattr_setpshared()` functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `pthread_condattr_setpshared()` function will fail if:

`EINVAL` The new value specified for the attribute is outside the range of legal values for that attribute.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_condattr_init(3THR) , pthread_create(3THR) , pthread_mutex_init(3THR) , pthread_cond_init(3THR) , attributes(5)

**NAME**

pthread_condattr_init, pthread_condattr_destroy – initialize or destroy condition variable attributes object

**SYNOPSIS**

cc -mt [ *flag...* ] *file*...- lpthread [ *library...* ]

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

**DESCRIPTION**

The function `pthread_condattr_init( )` initializes a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation.

At present, the only attribute available is the scope of condition variables. The default scope of the attribute is `PTHREAD_PROCESS_PRIVATE.`

Attempts to initialize previously initialized condition variable attributes object will leave the storage allocated by the previous initialization unallocated.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

The `pthread_condattr_destroy( )` function destroys a condition variable attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_condattr_destroy( )` to set the object referenced by *attr* to an invalid value. A destroyed condition variable attributes object can be re-initialized using `pthread_condattr_init( )` ; the results of otherwise referencing the object after it has been destroyed are undefined.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-dependent.

**RETURN VALUES**

If successful, the `pthread_condattr_init( )` and `pthread_condattr_destroy( )` functions return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_condattr_init( )` function will fail if:

| | |
|---|---|
| `ENOMEM` | Insufficient memory exists to initialize the condition variable attributes object. |

The `pthread_condattr_destroy( )` function may fail if:

| | |
|---|---|
| `EINVAL` | The value specified by *attr* is invalid. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_condattr_getpshared(3THR) , pthread_condattr_setpshared(3THR) , pthread_cond_init(3THR) , pthread_create(3THR) , pthread_mutex_init(3THR) , attributes(5)

**NAME**

pthread_cond_init, pthread_cond_destroy – initialize or destroy condition variables

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*- lpthread [ *library...* ]

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);
pthread_cond_t cond= PTHREAD_COND_INITIALIZER;
```

**DESCRIPTION**

The function `pthread_cond_init()` initializes the condition variable referenced by *cond* with attributes referenced by *attr* . If *attr* is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. See `pthread_condattr_init`(3THR) . Upon successful initialization, the state of the condition variable becomes initialized.

Attempting to initialize an already initialized. condition variable results in undefined behavior.

The function `pthread_cond_destroy()` destroys the given condition variable specified by *cond* ; the object becomes, in effect, uninitialized. An implementation may cause `pthread_cond_destroy()` to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can be re-initialized using `pthread_cond_init()` ; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro `PTHREAD_COND_INITIALIZER` can be used to initialize condition variables that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_cond_init()` with parameter *attr* specified as NULL, except that no error checks are performed.

**RETURN VALUES**

If successful, the `pthread_cond_init()` and `pthread_cond_destroy()` functions return `0` . Otherwise, an error number is returned to indicate the error. The `EBUSY` and `EINVAL` error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by *cond* .

**ERRORS**

The `pthread_cond_init()` function will fail if:

`EAGAIN` The system lacked the necessary resources (other than memory) to initialize another condition variable.

`ENOMEM` Insufficient memory exists to initialize the condition variable.

The `pthread_cond_init()` function may fail if:

`EBUSY` The implementation has detected an attempt to re-initialize the object referenced by *cond* , a previously initialized, but not yet destroyed, condition variable.

`EINVAL` The value specified by *attr* is invalid.

The `pthread_cond_destroy()` function may fail if:

`EBUSY` The implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced (for example, while being used in a `pthread_cond_wait()` or `pthread_cond_timedwait()` ) by another thread.

`EINVAL` The value specified by *cond* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`condition`(3THR) , `pthread_cond_signal`(3THR) , `pthread_cond_broadcast`(3THR) , `pthread_cond_wait`(3THR) , `pthread_cond_timedwait`(3THR) , `pthread_condattr_init`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

pthread_cond_signal, pthread_cond_broadcast – signal or broadcast a condition

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

**DESCRIPTION**

These two functions are used to unblock threads blocked on a condition variable.

The `pthread_cond_signal( )` call unblocks at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond* ).

The `pthread_cond_broadcast( )` call unblocks all threads currently blocked on the specified condition variable *cond* .

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a `pthread_cond_signal( )` or `pthread_cond_broadcast( )` returns from its call to `pthread_cond_wait( )` or `pthread_cond_timedwait( )` , the thread owns the mutex with which it called `pthread_cond_wait( )` or `pthread_cond_timedwait( )` . The thread(s) that are unblocked contend for the mutex according to the scheduling policy (if applicable), and as if each had called `pthread_mutex_lock( )` .

The `pthread_cond_signal( )` or `pthread_cond_broadcast( )` functions may be called by a thread whether or not it currently owns the mutex that threads calling `pthread_cond_wait( )` or `pthread_cond_timedwait( )` have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex is locked by the thread calling `pthread_cond_signal( )` or `pthread_cond_broadcast( )` .

The `pthread_cond_signal( )` and `pthread_cond_broadcast( )` functions have no effect if there are no threads currently blocked on *cond* .

**RETURN VALUES**

If successful, the `pthread_cond_signal( )` and `pthread_cond_broadcast( )` functions return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_cond_signal( )` and `pthread_cond_broadcast( )` function may fail if:

`EINVAL` The value *cond* does not refer to an initialized condition variable.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

condition(3THR) , pthread_cond_init(3THR) , pthread_cond_wait(3THR) , pthread_cond_timedwait(3THR) , attributes(5) , standards(5)

**NAME**

pthread_cond_wait, pthread_cond_timedwait – wait on a condition

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*- lpthread [ *library...* ]

#include <pthread.h>
int **pthread_cond_wait**(pthread_cond_t **cond*, pthread_mutex_t **mutex*);

int **pthread_cond_timedwait**(pthread_cond_t **cond*, pthread_mutex_t **mutex*, const struct timespec **abstime*);

**DESCRIPTION**

The `pthread_cond_wait()` and `pthread_cond_timedwait()` functions are used to block on a condition variable. They are called with *mutex* locked by the calling thread or undefined behaviour will result.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond* ; atomically here means "atomically with respect to access by another thread to the mutex and then the condition variable". That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to `pthread_cond_signal()` or `pthread_cond_broadcast()` in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex has been locked and is owned by the calling thread.

When using condition variables there is always a boolean predicate, an invariant, associated with each condition wait that must be true before the thread should proceed. Spurious wakeups from the `pthread_cond_wait()` or `pthread_cond_timedwait()` functions may occur. Since the return from `pthread_cond_wait()` or `pthread_cond_timedwait()` does not imply anything about the value of this predicate, the predicate should always be re-evaluated.

The order in which blocked threads are awakened by `pthread_cond_signal()` or `pthread_cond_broadcast()` is determined by the scheduling policy. See `pthreads`(3THR) .

The effect of using more than one mutex for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable will result in undefined behavior.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to `PTHREAD_CANCEL_DEFERRED,` a side effect of acting upon a cancellation request while in a condition wait is that the mutex is re-acquired before calling the first cancellation cleanup handler.

A thread that has been unblocked because it has been canceled while blocked in a call to `pthread_cond_wait()` or `pthread_cond_timedwait()` does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The `pthread_cond_timedwait( )` function is the same as `pthread_cond_wait( )` except that an error is returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds *abstime* ) before the condition *cond* is signaled or broadcasted, or if the absolute time specified by *abstime* has already been passed at the time of the call. When such time-outs occur, `pthread_cond_timedwait( )` will nonetheless release and reacquire the mutex referenced by *mutex* . The function `pthread_cond_timedwait( )` is also a cancellation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it returns `0` due to spurious wakeup.

**RETURN VALUES**

Except in the case of `ETIMEDOUT` , all these error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return, in effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable specified by *cond* .

Upon successful completion, a value of `0` is returned. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_cond_timedwait( )` function will fail if:

`ETIMEDOUT` The time specified by *abstime* to `pthread_cond_timedwait( )` has passed.

The `pthread_cond_wait( )` and `pthread_cond_timedwait( )` functions may fail if:

`EINVAL` The value specified by *cond* , *mutex* , or *abstime* is invalid.

`EINVAL` Different mutexes were supplied for concurrent `pthread_cond_wait( )` or `pthread_cond_timedwait( )` operations on the same condition variable.

`EINVAL` The mutex was not owned by the current thread at the time of the call.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`condition`(3THR) , `pthread_cond_signal`(3THR) , `pthread_cond_broadcast`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

pthread_create – create a thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
int **pthread_create**(pthread_t **thread*, const pthread_attr_t **attr*, void *(**start_routine*, void*),void **arg*);

**DESCRIPTION**

The `pthread_create( )` function is used to create a new thread, with attributes specified by *attr*, within a process. If *attr* is `NULL,` the default attributes are used. (See `pthread_attr_init`(3THR)). If the attributes specified by *attr* are modified later, the thread's attributes are not affected. Upon successful completion, `pthread_create( )` stores the ID of the created thread in the location referenced by *thread*.

The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect is as if there was an implicit call to `pthread_exit( )` using the return value of *start_routine* as the exit status. Note that the thread in which `main( )` was originally invoked differs from this. When it returns from `main( )`, the effect is as if there was an implicit call to `exit( )` using the return value of `main( )` as the exit status.

The signal state of the new thread is initialised as follows:

- The signal mask is inherited from the creating thread.
- The set of signals pending for the new thread is empty.

Default thread creation:

```
pthread_t tid;
void *start_func(void *), *arg;

pthread_create(&tid, NULL, start_func, arg);
```

This would have the same effect as:

```
pthread_attr_t attr;

pthread_attr_init(&attr); /* initialize attr with default attributes */
pthread_create(&tid, &attr, start_func, arg);
```

User-defined thread creation: To create a thread that is scheduled on a system-wide basis, use:

```
pthread_attr_init(&attr); /* initialize attr with default attributes */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); /* system-wide contention */
pthread_create(&tid, &attr, start_func, arg);
```

To customize the attributes for POSIX threads, see `pthread_attr_init`(3THR).

A new thread created with `pthread_create()` uses the stack specified by the *stackaddr* attribute, and the stack continues for the number of bytes specified by the *stacksize* attribute. By default, the stack size is 1 megabyte for 32-bit processes and 2 megabyte for 64-bit processes (see `pthread_attr_setstacksize`(3THR)). If the default is used for both the *stackaddr* and *stacksize* attributes, `pthread_create()` creates a stack for the new thread with at least 1 megabyte for 32-bit processes and 2 megabyte for 64-bit processes. (For customizing stack sizes, see NOTES).

If `pthread_create()` fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

**RETURN VALUES**

If successful, the `pthread_create()` function returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_create()` function will fail if:

`ENOMEM` The system lacked the necessary resources to create another thread.

`EINVAL` The value specified by *attr* is invalid.

`EPERM` The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

**EXAMPLES**

**EXAMPLE 1** This is an example of concurrency with multi-threading. Since POSIX threads and Solaris threads are fully compatible even within the same process, this example uses `pthread_create()` if you execute `a.out 0`, or `thr_create()` if you execute `a.out 1`.

Five threads are created that simultaneously perform a time-consuming function, `sleep(10)`. If the execution of this process is timed, the results will show that all five individual calls to sleep for ten-seconds completed in about ten seconds, even on a uniprocessor. If a single-threaded process calls `sleep(10)` five times, the execution time will be about 50-seconds.

The command-line to time this process is:

`/usr/bin/time a.out 0 (for POSIX threading)`

or

`/usr/bin/time a.out 1 (for Solaris threading)`

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic 3-lines for threads */
#include <pthread.h>
#include <thread.h>
```

```
#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);   /* thread routine */
int i;
thread_t tid[NUM_THREADS];      /* array of thread IDs */

int
main(int argc, char *argv[])
{
        if (argc == 1)  {
                printf("use 0 as arg1 to use pthread_create( )\n");
                printf("or use 1 as arg1 to use thr_create( )\n");
                return (1);
        }

        switch (*argv[1])  {
        case '0':  /* POSIX */
                for ( i = 0; i < NUM_THREADS; i++)
                        pthread_create(&tid[i], NULL, sleeping,
                            (void *)SLEEP_TIME);
                for ( i = 0; i < NUM_THREADS; i++)
                        pthread_join(tid[i], NULL);
                break;

        case '1':  /* Solaris */
                for ( i = 0; i < NUM_THREADS; i++)
                        thr_create(NULL, 0, sleeping, (void *)SLEEP_TIME, 0,
                            &tid[i]);
                while (thr_join(NULL, NULL, NULL) == 0)
                        ;
                break;
        }  /* switch */
        printf("main( ) reporting that all %d threads have terminated\n", i);
        return (0);
}  /* main */

void *
sleeping(void *arg)
{
        int sleep_time = (int)arg;
        printf("thread %d sleeping %d seconds ...\n", thr_self( ), sleep_time);
        sleep(sleep_time);
        printf("\nthread %d awakening\n", thr_self( ));
        return (NULL);
}
```

**EXAMPLE 2** If `main( )` had not waited for the completion of the other threads (using `pthread_join`(3THR) or `thr_join`(3THR)), it would have continued to process concurrently until it reached the end of its routine and the entire process would have exited prematurely (see `exit`(2)).

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`fork`(2), `sysconf`(3C), `pthread_attr_init`(3THR), `pthread_cancel`(3THR), `pthread_exit`(3THR), `pthread_join`(3THR), `attributes`(5), `standards`(5)

**NOTES**

MT application threads execute independently of each other, thus their relative behavior is unpredictable. Therefore, it is possible for the thread executing `main()` to finish before all other user application threads.

`pthread_join`(3THR), on the other hand, must specify the terminating thread (IDs) for which it will wait.

A user-specified stack size must be greater than the value `PTHREAD_STACK_MIN.` A minimum stack size may not accommodate the stack frame for the user thread function *start_func*. If a stack size is specified, it must accommodate *start_func* requirements and the functions that it may call in turn, in addition to the minimum requirement.

It is usually very difficult to determine the runtime stack requirements for a thread. `PTHREAD_STACK_MIN` specifies how much stack storage is required to execute a `NULL` *start_func*. The total runtime requirements for stack storage are dependent on the storage required to do runtime linking, the amount of storage required by library runtimes (as `printf()`) that your thread calls. Since these storage parameters are not known before the program runs, it is best to use default stacks. If you know your runtime requirements or decide to use stacks that are larger than the default, then it makes sense to specify your own stacks.

**NAME**

pthread_detach – detach a thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
int **pthread_detach**(pthread_t *thread*);

**DESCRIPTION**

The `pthread_detach( )` function is used to indicate to the implementation that storage for the thread *thread* can be reclaimed when that thread terminates. In other words, `pthread_detach( )` dynamically resets the *detachstate* attribute of the thread to `PTHREAD_CREATE_DETACHED.` After a successful call to this function, it would not be necessary to reclaim the thread using `pthread_join( )`. See `pthread_join`(3THR). If *thread* has not terminated, `pthread_detach( )` will not cause it to terminate. The effect of multiple `pthread_detach( )` calls on the same target thread is unspecified.

**RETURN VALUES**

If successful, `pthread_detach( )` returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_detach( )` function will fail if:

`EINVAL` The implementation has detected that the value specified by *thread* does not refer to a joinable thread.

`ESRCH` No thread could be found corresponding to that specified by the given thread ID.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_create`(3THR), `pthread_join`(3THR), `attributes`(5), `standards`(5)

**NAME**

pthread_equal – compare thread IDs

**SYNOPSIS**

cc –mt [ *flag...* ] *file...*– lpthread [ *library...* ]

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

**DESCRIPTION**

This function compares the thread IDs *t1* and *t2*.

**RETURN VALUES**

The `pthread_equal()` function returns a non-zero value if *t1* and *t2* are equal. Otherwise, `0` is returned.

If *t1* or *t2* is an invalid thread ID, the behavior is undefined.

**ERRORS**

No errors are defined.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_create`(3THR), `pthread_self`(3THR), `attributes`(5)

**NOTES**

Solaris thread IDs do not require an equivalent function because the `thread_t` structure is an unsigned int.

**NAME**

pthread_exit – terminate calling thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
void **pthread_exit**(void **value_ptr*);

**DESCRIPTION**

The `pthread_exit( )` function terminates the calling thread, in a similar way that `exit`(3C) terminates the calling process. If the thread is not detached, the exit status specified by *value_ptr* is made available to any successful join with the terminating thread. See `pthread_join`(3THR). Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions will be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling any `atexit( )` routines that may exist.

An implicit call to `pthread_exit( )` is made when a thread other than the thread in which `main( )` was first invoked returns from the start routine that was used to create it. The function's return value serves as the thread's exit status.

The behavior of `pthread_exit( )` is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `pthread_exit( )`.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the `pthread_exit( )` *value_ptr* parameter value.

The process exits with an exit status of `0` after the last thread has been terminated. The behavior is as if the implementation called `exit( )` with a `0` argument at thread termination time.

**RETURN VALUES**

The `pthread_exit( )` function cannot return to its caller.

**ERRORS**

No errors are defined.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

exit(3C), pthread_cancel(3THR), pthread_create(3THR), pthread_join(3THR), pthread_key_create(3THR), attributes(5), standards(5)

**NAME**

pthread_getconcurrency, pthread_setconcurrency – get or set level of concurrency

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*- lpthread [ *library...* ]

#include <pthread.h>
int **pthread_getconcurrency**(void);int pthread_setconcurrency(int *new_level*);

**DESCRIPTION**

Unbound threads in a process may or may not be required to be simultaneously active. By default, the threads implementation ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency.

The `pthread_setconcurrency( )` function allows an application to inform the threads implementation of its desired concurrency level, *new_level* . The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If *new_level* is `0` , it causes the implementation to maintain the concurrency level at its discretion as if `pthread_setconcurrency( )` was never called.

The `pthread_getconcurrency( )` function returns the value set by a previous call to the `pthread_setconcurrency( )` function. If the `pthread_setconcurrency( )` function was not previously called, this function returns `0` to indicate that the implementation is maintaining the concurrency level.

When an application calls `pthread_setconcurrency( )` it is informing the implementation of its desired concurrency level. The implementation uses this as a hint, not a requirement.

If an implementation does not support multiplexing of user threads on top of several kernel scheduled entities, the `pthread_setconcurrency( )` and `pthread_getconcurrency( )` functions will be provided for source code compatibility but they will have no effect when called. To maintain the function semantics, the *new_level* parameter will be saved when `pthread_setconcurrency( )` is called so that a subsequent call to `pthread_getconcurrency( )` returns the same value.

**RETURN VALUES**

If successful, the `pthread_setconcurrency( )` function returns `0` . Otherwise, an error number is returned to indicate the error.

The `pthread_getconcurrency( )` function always returns the concurrency level set by a previous call to `pthread_setconcurrency( )` . If the `pthread_setconcurrency( )` function has never been called, `pthread_getconcurrency( )` returns `0` .

**ERRORS**

The `pthread_setconcurrency( )` function will fail if:

`EINVAL` The value specified by *new_level* is negative.

`EAGAIN` The value specific by *new_level* would cause a system resource to be exceeded.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_create`(3THR) , `pthread_attr_init`(3THR) , `attributes`(5)

## NAME

pthread_getschedparam, pthread_setschedparam – access dynamic thread scheduling parameters

## SYNOPSIS

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

```
#include <pthread.h>
int pthread_getschedparam(pthread_t thread, int *policy, struct sched_param *param);

int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param
*param);
```

## DESCRIPTION

The `pthread_getschedparam( )` and `pthread_setschedparam( )` allow the scheduling policy and scheduling parameters of individual threads within a multi-threaded process to be retrieved and set. Supported policies are `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`. See `pthreads`(3THR) . For `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`, the affected scheduling parameter is the *sched_priority* member of the `sched_param` structure.

The `pthread_getschedparam( )` function retrieves the scheduling policy and scheduling parameters for the thread whose thread ID is given by *thread* and stores those values in *policy* and *param* , respectively. The priority value returned from `pthread_getschedparam( )` is the value specified by the most recent `pthread_setschedparam( )` or `pthread_create( )` call affecting the target thread, and reflects any temporary adjustments to its priority as a result of any priority inheritance or ceiling functions. The `pthread_setschedparam( )` function sets the scheduling policy and associated scheduling parameters for the thread whose thread ID is given by *thread* to the policy and associated parameters provided in *policy* and *param* , respectively.

If the `pthread_setschedparam( )` function fails, no scheduling parameters will be changed for the target thread.

## RETURN VALUES

If successful, the `pthread_getschedparam( )` and `pthread_setschedparam( )` functions return `0` . Otherwise, an error number is returned to indicate the error.

## ERRORS

The `pthread_getschedparam( )` function may fail if:

`ESRCH` The value specified by *thread* does not refer to a existing thread.

The `pthread_setschedparam( )` function may fail if:

`EINVAL` The value specified by *policy* or one of the scheduling parameters associated with the scheduling policy *policy* is invalid.

`EPERM` The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.

`ESRCH` The value specified by *thread* does not refer to a existing thread.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_attr_init`(3THR) , `pthreads`(3THR) , `sched_setparam`(3RT) , `sched_getparam`(3RT) , `sched_setscheduler`(3RT) , `sched_getscheduler`(3RT) `attributes`(5) , `standards`(5)

**NAME**

pthread_getspecific, pthread_setspecific – manage thread-specific data

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*- lpthread [ *library...* ]

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);

void *pthread_getspecific(pthread_key_t key);
```

**DESCRIPTION**

The `pthread_setspecific( )` function associates a thread-specific *value* with a *key* obtained by way of a previous call to `pthread_key_create( )` . Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The `pthread_getspecific( )` function returns the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling `pthread_setspecific( )` or `pthread_getspecific( )` with a *key* value not obtained from `pthread_key_create( )` or after *key* has been deleted with `pthread_key_delete( )` is undefined.

Both `pthread_setspecific( )` and `pthread_getspecific( )` may be called from a thread-specific data destructor function. However, calling `pthread_setspecific( )` from a destructor may result in lost storage or infinite loops.

**RETURN VALUES**

The `pthread_getspecific( )` function returns the thread-specific data value associated with the given *key* . If no thread-specific data value is associated with *key* , then the value `NULL` is returned.

Upon successful completion, the `pthread_setspecific( )` function returns `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_setspecific( )` function will fail if:

`ENOMEM` Insufficient memory exists to associate the value with the key.

The `pthread_setspecific( )` function may fail if:

`EINVAL` The key value is invalid.

The `pthread_getspecific( )` function does not return errors.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_key_create`(3THR) `attributes`(5) , `standards`(5)

**NAME**

pthread_join – wait for thread termination

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

**DESCRIPTION**

The `pthread_join( )` function suspends processing of the calling thread until the target *thread* completes. *thread* must be a member of the current process and it cannot be a detached or daemon thread. See `pthread_create`(3THR).

Several threads cannot wait for the same thread to complete; one thread will complete successfully and the others will terminate with an error of `ESRCH`. `pthread_join( )` will not block processing of the calling thread if the target *thread* has already terminated.

`pthread_join( )` returns successfully when the target *thread* terminates. If a `pthread_join( )` call returns successfully with a non-null *status* argument, the value passed to `pthread_exit`(3THR) by the terminating thread will be placed in the location referenced by *status*.

If the `pthread_join( )` calling thread is cancelled, then the target *thread* will remain joinable by `pthread_join( )`. However, the calling thread may set up a cancellation cleanup handler on *thread* prior to the join call, which may detach the target thread by calling `pthread_detach`(3THR). (See `pthread_detach`(3THR) and `pthread_cancel`(3THR).)

**RETURN VALUES**

If successful, the `pthread_join( )` function returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_join( )` function will fail if:

`EINVAL` The implementation has detected that the value specified by *thread* does not refer to a joinable thread.

`ESRCH` No thread could be found corresponding to that specified by the given thread ID.

The `pthread_join( )` function may fail if:

`EDEADLK` A recursive deadlock was detected, the value of *thread* specifies the calling thread.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`wait`(2), `pthread_create`(3THR), `attributes`(5), `standards`(5)

**NOTES**

`pthread_join`(3THR), must specify the *thread* ID for whose termination it will wait.

Calling `pthread_join()` also "detaches" the thread, that is, `pthread_join()` includes the effect of `pthread_detach()`. Hence, if a thread were to be cancelled when blocked in `pthread_join()`, an explicit detach would have to be done in the cancellation cleanup handler. In fact, the routine `pthread_detach()` exists mainly for this reason.

**NAME**

pthread_key_create – create thread-specific data key

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>

int **pthread_key_create**(pthread_key_t **key*, void (**destructor*, void*));); 

**DESCRIPTION**

This function creates a thread-specific data key visible to all threads in the process. Key values provided by `pthread_key_create()` are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by `pthread_setspecific()` are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value `NULL` is associated with the new key in all active threads. Upon thread creation, the value `NULL` is associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-`NULL` destructor pointer, and the thread has a non-`NULL` value associated with that key, the function pointed to is called with the current associated value as its sole argument. Destructors can be called in any order.

If, after all the destructors have been called for all keys with non-`NULL` values, there are still some keys with non-`NULL` values, the process will be repeated. If, after at least `PTHREAD_DESTRUCTOR_ITERATIONS` iterations of destructor calls for outstanding non-`NULL` values, there are still some keys with non-`NULL` values, the process is continued, even though this might result in an infinite loop.

**RETURN VALUES**

If successful, the `pthread_key_create()` function stores the newly created key value at **key* and returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_key_create()` function will fail if:

`EAGAIN` The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process `PTHREAD_KEYS_MAX` has been exceeded.

`ENOMEM` Insufficient memory exists to create the key.

The `pthread_key_create()` function will not return an error code of `EINTR`.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_getspecific(3THR), pthread_setspecific(3THR), pthread_key_delete(3THR), attributes(5), standards(5)

**NAME**

pthread_key_delete – delete thread-specific data key

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
```

**DESCRIPTION**

This function deletes a thread-specific data key previously returned by `pthread_key_create()`. The thread-specific data values associated with *key* need not be `NULL` at the time `pthread_key_delete()` is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after `pthread_key_delete()` is called. Any attempt to use *key* following the call to `pthread_key_delete()` results in undefined behaviour.

The `pthread_key_delete()` function is callable from within destructor functions. No destructor functions will be invoked by `pthread_key_delete()`. Any destructor function that may have been associated with *key* will no longer be called upon thread exit.

**RETURN VALUES**

If successful, the `pthread_key_delete()` function returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_key_delete()` function may fail if:

`EINVAL` The *key* value is invalid.

The `pthread_key_delete()` function will not return an error code of `EINTR`.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_key_create`(3THR), `attributes`(5), `standards`(5)

**NAME**

pthread_kill – send a signal to a thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

```
#include <signal.h>
#include <pthread.h>
int pthread_kill(pthread_t thread, int sig);
```

**DESCRIPTION**

The `pthread_kill()` function is used to request that a signal be delivered to the specified thread.

As in `kill()`, if *sig* is `0`, error checking is performed but no signal is actually sent.

**RETURN VALUES**

Upon successful completion, the function returns a value of `0`. Otherwise the function returns an error number. If the `pthread_kill()` function fails, no signal is sent.

**ERRORS**

The `pthread_kill()` function will fail if:

`ESRCH` No thread could be found corresponding to that specified by the given thread ID.

`EINVAL` The value of the *sig* argument is an invalid or unsupported signal number.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`kill`(1), `pthread_self`(3THR), `pthread_sigmask`(3THR), `raise`(3C), `attributes`(5), `standards`(5)

## NAME

pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling – get and set prioceiling attribute of mutex attribute object

## SYNOPSIS

```
cc -mt [ flag... ] file... -lpthread [ library... ]
#include <pthread.h>
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling
int *oldceiling);

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr, int
*prioceiling);
```

## DESCRIPTION

The `pthread_mutexattr_getprioceiling( )` and `pthread_mutexattr_setprioceiling( )` functions, respectively, get and set the priority ceiling attribute of a mutex attribute object pointed to by *attr* , which was previously created by the `pthread_mutexattr_init( )` function.

The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of *prioceiling* must be within the maximum range of priorities defined by `SCHED_FIFO` .

The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex must be set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* must be within the maximum range of priorities defined under the `SCHED_FIFO` scheduling policy.

The ceiling value should be drawn from the range of priorities for the `SCHED_FIFO` policy. When a thread acquires such a mutex, the policy of the thread at mutex acquisition should match that from which the ceiling value was derived (`SCHED_FIFO` , in this case). If a thread changes its scheduling policy while holding a ceiling mutex, the behavior of `pthread_mutex_lock( )` and `pthread_mutex_unlock( )` on this mutex is undefined. See `pthread_mutex_lock`(3THR) .

The ceiling value should not be treated as a persistent value resident in a `pthread_mutex_t` that is valid across upgrades of Solaris. The semantics of the actual ceiling value are determined by the existing priority range for the `SCHED_FIFO` policy, as returned by the `sched_get_priority_min( )` and `sched_get_priority_max( )` functions (see `sched_get_priority_min`(3RT) ) when called on the version of Solaris on which the ceiling value is being utilized.

## RETURN VALUES

Upon successful completion, the `pthread_mutexattr_getprioceiling( )` and `pthread_mutexattr_setprioceiling( )` functions return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions will fail if:

`ENOSYS` The `_POSIX_THREAD_PRIO_PROTECT` option is not defined and the system does not support the function.

The `pthread_mutexattr_getprioceiling()` and `pthread_mutexattr_setprioceiling()` functions may fail if:

`EINVAL` The value specified by *attr* or *prioceiling* is invalid.

`EPERM` The caller does not have the privilege to perform the operation.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_cond_init`(3THR) , `pthread_create`(3THR) , `pthread_mutex_init`(3THR) , `pthread_mutex_lock`(3THR) , `sched_get_priority_min`(3RT) , `attributes`(5) , `standards`(5)

## NAME

pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol – get and set protocol attribute of mutex attribute object

## SYNOPSIS

cc -mt [ *flag*... ] *file*... -lpthread [ *library*... ]
#include <pthread.h>
int **pthread_mutexattr_getprotocol**(const pthread_mutexattr_t **attr*, int **protocol*);

int **pthread_mutexattr_setprotocol**(pthread_mutexattr_t **attr*, int *protocol*);

## DESCRIPTION

The `pthread_mutexattr_setprotocol()` and `pthread_mutexattr_getprotocol()` functions, respectively, set and get the protocol attribute of a mutex attribute object pointed to by *attr* , which was previously created by the `pthread_mutexattr_init()` function.

The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of *protocol* may be one of `PTHREAD_PRIO_NONE` , `PTHREAD_PRIO_INHERIT` , or `PTHREAD_PRIO_PROTECT` , which are defined by the header `<pthread.h>` .

When a thread owns a mutex with the `PTHREAD_PRIO_NONE` protocol attribute, its priority and scheduling are not affected by its mutex ownership.

When a thread is blocking higher priority threads because of owning one or more mutexes with the `PTHREAD_PRIO_INHERIT` protocol attribute, it executes at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

When a thread owns one or more mutexes initialized with the `PTHREAD_PRIO_PROTECT` protocol, it executes at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes.

While a thread is holding a mutex that has been initialized with the `PRIO_INHERIT` or `PRIO_PROTECT` protocol attributes, it will not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed, such as by a call to `sched_setparam()` . Likewise, when a thread unlocks a mutex that has been initialized with the `PRIO_INHERIT` or `PRIO_PROTECT` protocol attributes, it will not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed.

If a thread simultaneously owns several mutexes initialized with different protocols, it will execute at the highest of the priorities that it would have obtained by each of these protocols.

When a thread makes a call to `pthread_mutex_lock()` , if the symbol `_POSIX_THREAD_PRIO_INHERIT` is defined and the mutex was initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT , when the calling thread is blocked because the mutex is owned by another thread, that owner thread will inherit the priority level of the calling thread as long as it continues to own the mutex. The implementation updates its execution priority to the maximum of its assigned priority and all its inherited priorities. Furthermore, if this owner thread becomes blocked on another mutex, the same priority inheritance effect will be propagated to the other owner thread, in a recursive manner.

If the symbol _POSIX_THREAD_PRIO_INHERIT is defined, when a mutex initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT dies, the behavior depends on the robustness attribute of the mutex. See pthread_mutexattr_getrobust_np(3THR) .

A thread that uses mutexes initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT *protocol* attribute values should have its *contentionscope* attribute equal to PTHREAD_SCOPE_SYSTEM (see pthread_attr_getscope(3THR) ) and its scheduling policy equal to SCHED_FIFO or SCHED_RR (see pthread_attr_getschedparam(3THR) and pthread_getschedparam(3THR) ).

If a thread with *contentionscope* attribute equal to PTHREAD_SCOPE_PROCESS and / or its scheduling policy equal to SCHED_OTHER uses a mutex initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT *protocol* attribute value, the effect on the thread's scheduling and priority is unspecified.

The _POSIX_THREAD_PRIO_INHERIT and _POSIX_THREAD_PRIO_PROTECT options are designed to provide features to solve priority inversion due to mutexes. A priority inheritance or priority ceiling mutex is designed to minimize the dispatch latency of a high priority thread when a low priority thread is holding a mutex required by the high priority thread. This is a specific need for the realtime application domain.

Threads created by realtime applications need to be such that their priorities can influence their access to system resources (CPU resources, at least), in competition with all threads running on the system.

Threads that use priority inheritance or priority ceiling locks should be in the PTHREAD_SCOPE_SYSTEM (SYSTEM for short) scheduling contention scope (or bound threads), which are defined as threads that compete with threads across the system and across different processes.

Threads in the PTHREAD_SCOPE_PROCESS (PROCESS for short) scheduling contention scope (or unbound threads) do not compete with threads in other processes, making them unsuitable for the needs of the realtime application domain. Therefore, only bound threads should be used with priority inheritance and priority ceiling mutexes. In addition, the scheduling policies for these threads should be either SCHED_FIFO or SCHED_RR (the realtime scheduling policies).

**RETURN VALUES**

Upon successful completion, the pthread_mutexattr_getprotocol( ) and pthread_mutexattr_setprotocol( ) functions return 0 . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The pthread_mutexattr_getprotocol( ) and pthread_mutexattr_setprotocol( ) functions will fail if:

ENOSYS — Neither of the options _POSIX_THREAD_PRIO_PROTECT and _POSIX_THREAD_PRIO_INHERIT is defined and the system does not support the function.

ENOTSUP — The value specified by *protocol* is an unsupported value.

The pthread_mutexattr_getprotocol( ) and pthread_mutexattr_setprotocol( ) functions may fail if:

EINVAL — The value specified by *attr* or *protocol* is invalid.

EPERM — The caller does not have the privilege to perform the operation.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_create(3THR) , pthread_mutex_init(3THR) , pthread_cond_init(3THR) , pthread_mutexattr_getrobust_np(3THR) , attributes(5) , standards(5)

**NAME**

pthread_mutexattr_getpshared, pthread_mutexattr_setpshared – get and set process-shared attribute

**SYNOPSIS**

cc -mt [ *flag*... ] *file*... -lpthread [ *library*... ]
#include <pthread.h>
int **pthread_mutexattr_getpshared**(const pthread_mutexattr_t **attr*, int **pshared*);

int **pthread_mutexattr_setpshared**(pthread_mutexattr_t **attr*, int *pshared*);

**DESCRIPTION**

The `pthread_mutexattr_getpshared( )` function obtains the value of the *process-shared* attribute from the attributes object referenced by *attr* . The `pthread_mutexattr_setpshared( )` function is used to set the *process-shared* attribute in an initialized attributes object referenced by *attr* .

The *process-shared* attribute is set to `PTHREAD_PROCESS_SHARED` to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE` , the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE` .

**RETURN VALUES**

Upon successful completion, `pthread_mutexattr_getpshared( )` returns `0` and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number is returned to indicate the error.

Upon successful completion, `pthread_mutexattr_setpshared( )` returns `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_mutexattr_getpshared( )` and `pthread_mutexattr_setpshared( )` functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

The `pthread_mutexattr_setpshared( )` function may fail if:

`EINVAL` The new value specified for the attribute is outside the range of legal values for that attribute.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO** pthread_create(3THR) , pthread_mutex_init(3THR) , pthread_mutexattr_init(3THR) , pthread_cond_init(3THR) , attributes(5) , standards(5)

## NAME

pthread_mutexattr_getrobust_np, pthread_mutexattr_setrobust_np – get or set robustness attribute of mutex attribute object

## SYNOPSIS

cc -mt [ *flag*... ] *file*... -lpthread [ *library*... ]
#include <pthread.h>
**int pthread_mutexattr_getrobust_np**(const pthread_mutexattr_t **attr*, int **robustness*);

**int pthread_mutexattr_setrobust_np**(pthread_mutexattr_t **attr*, int *robustness*);

## DESCRIPTION

The following applies only if the symbol _POSIX_THREAD_PRIO_INHERIT is defined, and the mutex attributes object *attr* should be used only to initialize mutexes that will also be initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT . See pthread_mutexattr_getprotocol(3THR) .

The pthread_mutexattr_setrobust_np( ) and pthread_mutexattr_getrobust_np( ) functions set and get the *robustness* attribute of a mutex attribute object pointed to by *attr* that was previously created by the function pthread_mutexattr_init(3THR) .

The *robustness* attribute defines the behavior when the owner of a mutex dies. The value of *robustness* may be ether PTHREAD_MUTEX_ROBUST_NP or PTHREAD_MUTEX_STALLED_NP , which are defined by the header <pthread.h> . The default value of the *robustness* attribute is PTHREAD_MUTEX_STALLED_NP .

When the owner of a mutex with the PTHREAD_MUTEX_STALLED_NP *robustness* attribute dies, all future calls to pthread_mutex_lock(3THR) for this mutex will be blocked from progress in an unspecified manner.

When the owner of a mutex with the PTHREAD_MUTEX_ROBUST_NP *robustness* attribute dies, the mutex is unlocked. The next owner of this mutex acquires it with an error value of EOWNERDEAD . Note that the application must always check the return value from pthread_mutex_lock( ) for a mutex initialized with the PTHREAD_MUTEX_ROBUST_NP *robustness* attribute. The new owner of this mutex should then attempt to make the state protected by the mutex consistent, since this state could have been left inconsistent when the last owner died. If the new owner is able to make the state consistent, it should call pthread_mutex_consistent_np(3THR) for the mutex and then unlock the mutex. If for any reason the new owner is not able to make the state consistent, it should not call pthread_mutex_consistent_np( ) for the mutex, but should simply unlock the mutex. In the latter scenario, all waiters will be awakened and all subsequent calls to pthread_mutex_lock( ) will fail in acquiring the mutex with an error value of ENOTRECOVERABLE . The mutex can then be made consistent by uninitializing the mutex with the pthread_mutex_destroy( ) function and reinitializing it with the pthread_mutex_init( ) function. If the thread that acquired the lock with EOWNERDEAD dies, the next owner will acquire the lock with an error value of EOWNERDEAD .

Note that the mutex may be in memory shared between processes or in memory private to a process, i.e. the "owner" referenced above is a thread, either within or outside the requestor's process.

The mutex memory must be zeroed before initialization.

**RETURN VALUES**

Upon successful completion, the pthread_mutexattr_getrobust_np( ) and pthread_mutexattr_setrobust_np( ) functions return 0 . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The pthread_mutexattr_getrobust_np( ) and pthread_mutexattr_setrobust_np( ) functions will fail if:

EINVAL The value specified by *attr* or *robustness* is invalid.

ENOSYS The option _POSIX_THREAD_PRIO_INHERIT is not defined and the implementation does not support the function.

ENOTSUP The value specified by robustness is an unsupported value.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

mutex(3THR) , pthread_mutex_lock(3THR) , pthread_mutex_consistent_np(3THR) , pthread_mutexattr_getprotocol(3THR) , attributes(5) , standards(5)

**NAME**

pthread_mutexattr_gettype, pthread_mutexattr_settype – get or set a mutex type

**SYNOPSIS**

cc -mt [ *flag*... ] *file*... -lpthread [ *library*... ]
#include <pthread.h>
int **pthread_mutexattr_gettype**(pthread_mutexattr_t **attr*, int **type*);

int **pthread_mutexattr_settype**(pthread_mutexattr_t **attr*, int *type*);

**DESCRIPTION**

The pthread_mutexattr_gettype( ) and pthread_mutexattr_settype( ) functions respectively get and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT .

The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types include:

PTHREAD_MUTEX_NORMAL — This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK — This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex that another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.

PTHREAD_MUTEX_RECURSIVE — A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock that can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire

| | |
|---|---|
| | the mutex. A thread attempting to unlock a mutex that another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error. This type of mutex is only supported for mutexes whose process shared attribute is `PTHREAD_PROCESS_PRIVATE`. |
| `PTHREAD_MUTEX_DEFAULT` | Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex of this type that was not locked by the calling thread results in undefined behavior. Attempting to unlock a mutex of this type that is not locked results in undefined behavior. An implementation is allowed to map this mutex to one of the other mutex types. |

**RETURN VALUES**

Upon successful completion, the `pthread_mutexattr_settype()` function returns `0`. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_mutexattr_gettype()` function returns `0` and stores the value of the *type* attribute of *attr* in the object referenced by the *type* parameter. Otherwise an error number is returned to indicate the error.

**ERRORS**

The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions will fail if:

`EINVAL` The value `type` is invalid.

The `pthread_mutexattr_gettype()` and `pthread_mutexattr_settype()` functions may fail if:

`EINVAL` The value specified by *attr* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO** `pthread_cond_timedwait`(3THR) , `pthread_cond_wait`(3THR) , `attributes`(5)

**NOTES** It is advised that an application should not use a `PTHREAD_MUTEX_RECURSIVE` mutex with condition variables `PTHREAD_MUTEX_RECURSIVE` because the implicit unlock performed for a `pthread_cond_wait()` or `pthread_cond_timedwait()` will not actually release the mutex (if it had been locked multiple times). If this occurs, no other thread can satisfy the condition of the predicate.

**NAME**

pthread_mutexattr_init, pthread_mutexattr_destroy – initialize and destroy mutex attributes object

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]
#include <pthread.h>
int **pthread_mutexattr_init**(pthread_mutexattr_t **attr*);

int **pthread_mutexattr_destroy**(pthread_mutexattr_t **attr*);

**DESCRIPTION**

The `pthread_mutexattr_init()` function initializes a mutex attributes object *attr* with the default value for all of the attributes defined by the implementation.

The effect of initializing an already initialized mutex attributes object is undefined.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) does not affect any previously initialized mutexes.

The `pthread_mutexattr_destroy()` function destroys a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause `pthread_mutexattr_destroy()` to set the object referenced by *attr* to an invalid value. A destroyed mutex attributes object can be re-initialized using `pthread_mutexattr_init()` ; the results of otherwise referencing the object after it has been destroyed are undefined.

**RETURN VALUES**

Upon successful completion, `pthread_mutexattr_init()` and `pthread_mutexattr_destroy()` return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_mutexattr_init()` function may fail if:

`ENOMEM` Insufficient memory exists to initialize the mutex attributes object.

The `pthread_mutexattr_destroy()` function may fail if:

`EINVAL` The value specified by *attr* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_create`(3THR) , `pthread_mutex_init`(3THR) , `pthread_mutexattr_init`(3THR) , `pthread_cond_init`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

pthread_mutex_consistent_np – make a mutex consistent after owner death

**SYNOPSIS**

cc –mt [ *flag*... ] *file*... –lpthread [ *library*... ]
#include <pthread.h>
**int pthread_mutex_consistent_np**(pthread_mutex_t **mutex*);

**DESCRIPTION**

The following applies only if the symbol `_POSIX_THREAD_PRIO_INHERIT` is defined, and for mutexes that have been initialized with the protocol attribute having the value `PTHREAD_PRIO_INHERIT`. See `pthread_mutexattr_getprotocol`(3THR).

The mutex object referenced by *mutex* is made consistent by calling `pthread_mutex_consistent_np( )`.

A consistent mutex becomes inconsistent and is unlocked if its owner dies while holding it. A subsequent owner of the mutex will acquire the mutex with `pthread_mutex_lock`(3THR), which will return `EOWNERDEAD` to indicate that the acquired mutex is inconsistent.

The `pthread_mutex_consistent_np( )` function should be called while holding the mutex acquired by a previous call to `pthread_mutex_lock( )`that returned `EOWNERDEAD`.

Since the critical section protected by the mutex could have been left in an inconsistent state by the dead owner, the caller should make the mutex consistent only if it is able to make the critical section protected by the mutex consistent.

Calls to `pthread_mutex_lock( )`, `pthread_mutex_unlock( )`, and `pthread_mutex_trylock( )` for a consistent mutex will behave in the normal manner.

The behavior of `pthread_mutex_consistent_np( )` for a mutex which is not inconsistent, or which is not held, is undefined.

**RETURN VALUES**

Upon successful completion, the `pthread_mutexattr_consistent_np( )` function returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_mutex_consistent_np( )` function will fail if:

`ENOSYS` The option `_POSIX_THREAD_PRIO_INHERIT` is not defined and the implementation does not support the function.

The `pthread_mutex_consistent_np( )` function may fail if:

`EINVAL` The value specified by *mutex* is invalid, or the mutex does not have the appropriate attributes.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

mutex(3THR), pthread_mutex_lock(3THR), pthread_mutexattr_getprotocol(3THR), pthread_mutexattr_getrobust_np(3THR), attributes(5), standards(5)

**NAME**

pthread_mutex_getprioceiling, pthread_mutex_setprioceiling – change the priority ceiling of a mutex

**SYNOPSIS**

cc -mt [ *flag*... ] *file*... -lpthread [ *library*... ]
#include <pthread.h>
int **pthread_mutex_setprioceiling**(pthread_mutex_t **mutex*, int *prioceiling*, int **old_ceiling*);

int **pthread_mutex_getprioceiling**(const pthread_mutex_t **mutex*, int **prioceiling*);

**DESCRIPTION**

The `pthread_mutex_getprioceiling()` function returns the current priority ceiling of the mutex.

The `pthread_mutex_setprioceiling()` function either locks the mutex if it is unlocked, or blocks until it can successfully lock the mutex, then it changes the mutex's priority ceiling and releases the mutex. When the change is successful, the previous value of the priority ceiling is returned in *old_ceiling* . The process of locking the mutex need not adhere to the priority protect protocol.

If the `pthread_mutex_setprioceiling()` function fails, the mutex priority ceiling is not changed.

The ceiling value should be drawn from the range of priorities for the `SCHED_FIFO` policy. When a thread acquires such a mutex, the policy of the thread at mutex acquisition should match that from which the ceiling value was derived (`SCHED_FIFO` , in this case). If a thread changes its scheduling policy while holding a ceiling mutex, the behavior of `pthread_mutex_lock()` and `pthread_mutex_unlock()` on this mutex is undefined. See `pthread_mutex_lock`(3THR) .

The ceiling value should not be treated as a persistent value resident in a `pthread_mutex_t` that is valid across upgrades of Solaris. The semantics of the actual ceiling value are determined by the existing priority range for the `SCHED_FIFO` policy, as returned by the `sched_get_priority_min()` and `sched_get_priority_max()` functions (see `sched_get_priority_min`(3RT) ) when called on the version of Solaris on which the ceiling value is being utilized.

**RETURN VALUES**

Upon successful completion, the `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions return `0` . Otherwise, an error number is returned to indicate the error.

These functions are not currently supported and will always return `ENOSYS.`

**ERRORS**

The `pthread_mutex_getprioceiling()` and `pthread_mutex_setprioceiling()` functions will fail if:

ENOSYS — The option _POSIX_THREAD_PRIO_PROTECT is not defined and the system does not support the function.

The pthread_mutex_setprioceiling( ) function will fail if:

EINVAL — The mutex was not initialized with its *protocol* attribute having the value of PTHREAD_PRIO_PROTECT .

The pthread_mutex_getprioceiling( ) and pthread_mutex_setprioceiling( ) functions may fail if:

EINVAL — The priority requested by *prioceiling* is out of range.

EINVAL — The value specified by *mutex* does not refer to a currently existing mutex.

ENOSYS — The system does not support the priority ceiling protocol for mutexes.

EPERM — The caller does not have the privilege to perform the operation.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_mutex_init(3THR) , pthread_mutex_lock(3THR) , sched_get_priority_min(3RT) attributes(5) , standards(5)

**NAME**

pthread_mutex_init, pthread_mutex_destroy – initialize or destroy a mutex

**SYNOPSIS**

cc -mt [ *flag*... ] *file*... -lpthread [ *library*... ]
#include <pthread.h>
int **pthread_mutex_init**(pthread_mutex_t **mutex*, const pthread_mutexattr_t **attr*);

int **pthread_mutex_destroy**(pthread_mutex_t **mutex*);
pthread_mutex_t *mutex*= PTHREAD_MUTEX_INITIALIZER

**DESCRIPTION**

The `pthread_mutex_init()` function initializes the mutex referenced by *mutex* with attributes specified by *attr* . If *attr* is `NULL` , the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Attempting to initialize an already initialized mutex results in undefined behavior.

The `pthread_mutex_destroy()` function destroys the mutex object referenced by *mutex* ; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be re-initialized using `pthread_mutex_init()` ; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize mutexes that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_mutex_init()` with parameter *attr* specified as `NULL` , except that no error checks are performed.

**RETURN VALUES**

If successful, the `pthread_mutex_init()` and `pthread_mutex_destroy()` functions return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_mutex_init()` function will fail if:

`EAGAIN` The system lacked the necessary resources (other than memory) to initialize another mutex.

`ENOMEM` Insufficient memory exists to initialize the mutex.

`EPERM` The caller does not have the privilege to perform the operation.

The `pthread_mutex_init()` function may fail if:

`EBUSY` An attempt was detected to re-initialize the object referenced by *mutex* , a mutex previously initialized but not yet destroyed.

`EINVAL` The value specified by *attr* or *mutex* is invalid.

The `pthread_mutex_destroy( )` function may fail if:

`EBUSY` An attempt was detected to destroy the object referenced by *mutex* while it is locked or referenced (for example, while being used in a `pthread_cond_wait`(3THR) or `pthread_cond_timedwait`(3THR) ) by another thread.

`EINVAL` The value specified by *mutex* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`mutex`(3THR) , `pthread_cond_timedwait`(3THR) , `pthread_cond_wait`(3THR) , `pthread_mutex_getprioceiling`(3THR) , `pthread_mutex_lock`(3THR) , `pthread_mutex_unlock`(3THR) , `pthread_mutex_setprioceiling`(3THR) , `pthread_mutex_trylock`(3THR) , `pthread_mutexattr_getpshared`(3THR) , `pthread_mutexattr_setpshared`(3THR) `attributes`(5) , `standards`(5)

**NAME**

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock – lock or unlock a mutex

**SYNOPSIS**

cc -mt [ *flag*... ] *file*... -lpthread [ *library*... ]
#include <pthread.h>
int **pthread_mutex_lock**(pthread_mutex_t **mutex*);

int **pthread_mutex_trylock**(pthread_mutex_t **mutex*);

int **pthread_mutex_unlock**(pthread_mutex_t **mutex*);

**DESCRIPTION**

The mutex object referenced by mutex is locked by calling `pthread_mutex_lock()` . If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

If the mutex type is `PTHREAD_MUTEX_NORMAL` , deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, undefined behavior results.

If the mutex type is `PTHREAD_MUTEX_ERRORCHECK` , then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_RECURSIVE` , then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to 1. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches `0` , the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, an error will be returned.

If the mutex type is `PTHREAD_MUTEX_DEFAULT` , attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The `pthread_mutex_trylock()` function is identical to `pthread_mutex_lock()` except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call returns immediately.

The `pthread_mutex_unlock()` function releases the mutex object referenced by *mutex* . The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when `pthread_mutex_unlock()` is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex becomes available when the count reaches 0 and the calling thread no longer has any locks on this mutex.)

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

**RETURN VALUES**

If successful, the pthread_mutex_lock( ) and pthread_mutex_unlock( ) functions return 0 . Otherwise, an error number is returned to indicate the error.

The pthread_mutex_trylock( ) function returns 0 if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The pthread_mutex_lock( ) and pthread_mutex_trylock( ) functions will fail if:

EINVAL The *mutex* was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than the mutex's current priority ceiling.

The pthread_mutex_trylock( ) function will fail if:

EBUSY The *mutex* could not be acquired because it was already locked.

The pthread_mutex_lock( ) , pthread_mutex_trylock( ) and pthread_mutex_unlock( ) functions may fail if:

EINVAL The value specified by *mutex* does not refer to an initialized mutex object.

EAGAIN The mutex could not be acquired because the maximum number of recursive locks for *mutex* has been exceeded.

The pthread_mutex_lock( ) function may fail if:

EDEADLK The current thread already owns the mutex.

The pthread_mutex_unlock( ) function may fail if:

EPERM The current thread does not own the mutex.

When a thread makes a call to pthread_mutex_lock( ) or pthread_mutex_trylock( ) , if the symbol _POSIX_THREAD_PRIO_INHERIT is defined and the mutex is initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT and the robustness attribute having the value PTHREAD_MUTEX_ROBUST_NP (see pthread_mutexattr_getrobust_np(3THR) ), the pthread_mutex_lock( ) and pthread_mutex_trylock( ) functions will fail if:

EOWNERDEAD
: The last owner of this mutex died while holding the mutex. This mutex is now owned by the caller. The caller must now attempt to make the state protected by the mutex consistent. If it is able to clean up the state, then it should call pthread_mutex_consistent_np( ) for the mutex and unlock the mutex. Subsequent calls to pthread_mutex_lock( ) and pthread_mutex_trylock( ) will behave normally, as before. If the caller is not able to clean up the state, pthread_mutex_consistent_np( ) should not be called for the mutex, but it should be unlocked. Subsequent calls to pthread_mutex_lock( ) and pthread_mutex_trylock( ) will fail to acquire the mutex with the error value ENOTRECOVERABLE . If the owner who acquired the lock with EOWNERDEAD dies, the next owner will acquire the lock with EOWNERDEAD .

ENOTRECOVERABLE
: The mutex trying to be acquired is protecting the state that has been left irrecoverable by the mutex's last owner, who died while holding the lock. The mutex has not been acquired. This condition can occur when the lock was previously acquired with EOWNERDEAD , and the owner was not able to clean up the state and unlocked the mutex without making the mutex consistent.

ENOMEM
: The limit on the number of simultaneously held mutexes has been exceeded.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_mutex_init(3THR) , pthread_mutex_destroy(3THR) , pthread_mutex_consistent_np(3THR) , pthread_mutexattr_getrobust_np(3THR) , attributes(5) , standards(5)

**NOTES**

In the current implementation of threads, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `mutex_lock()`, `mutex_unlock()`, `pthread_mutex_trylock()`, and `mutex_trylock()` do not validate the mutex type. Therefore, an uninitialized mutex or a mutex with an invalid type does not return `EINVAL`. Interfaces for mutexes with an invalid type have unspecified behavior.

Uninitialized mutexes that are allocated locally may contain junk data. Such mutexes need to be initialized using `pthread_mutex_init()` or `mutex_init()`.

**NAME**

pthread_once – initialize dynamic package

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
pthread_once_t *once_control* = PTHREAD_ONCE_INIT;
int **pthread_once**(pthread_once_t **once_control*, void (**init_routine*, void));

**DESCRIPTION**

If any thread in a process with a *once_control* parameter makes a call to `pthread_once( )`, the first call will summon the `init_routine( )`, but subsequent calls will not. The *once_control* parameter determines whether the associated initialization routine has been called. The `init_routine( )` is complete upon return of `pthread_once( )`.

`pthread_once( )` is not a cancellation point; however, if the function `init_routine( )` is a cancellation point and is canceled, the effect on *once_control* is the same as if `pthread_once( )` had never been called.

The constant `PTHREAD_ONCE_INIT` is defined in the `<pthread.h>` header.

If *once_control* has automatic storage duration or is not initialized by `PTHREAD_ONCE_INIT`, the behavior of `pthread_once( )` is undefined.

**RETURN VALUES**

Upon successful completion, `pthread_once( )` returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

`EINVAL` *once_control* or *init_routine* is `NULL`.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`attributes`(5)

**NOTES**

Solaris threads do not offer this functionality.

## NAME

pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared – get or set process-shared attribute of read-write lock attributes object

## SYNOPSIS

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

```
#include <pthread.h>
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *pshared);

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

## DESCRIPTION

The *process-shared* attribute is set to `PTHREAD_PROCESS_SHARED` to permit a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE,` the read-write lock will only be operated upon by threads created within the same process as the thread that initialised the read-write lock; if threads of differing processes attempt to operate on such a read-write lock, the behaviour is undefined. The default value of the *process-shared* attribute is `PTHREAD_PROCESS_PRIVATE.`

The `pthread_rwlockattr_getpshared( )` function obtains the value of the *process-shared* attribute from the initialised attributes object referenced by *attr* . The `pthread_rwlockattr_setpshared( )` function is used to set the *process-shared* attribute in an initialised attributes object referenced by attr.

## RETURN VALUES

If successful, the `pthread_rwlockattr_setpshared( )` function returns `0` . Otherwise, an error number is returned to indicate the error.

Upon successful completion, the `pthread_rwlockattr_getpshared( )` returns `0` and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise an error number is returned to indicate the error.

## ERRORS

The `pthread_rwlockattr_getpshared( )` and `pthread_rwlockattr_setpshared( )` functions will fail if:

`EINVAL` The value specified by *attr* or *pshared* is invalid.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`pthread_rwlock_init`(3THR) , `pthread_rwlock_rdlock`(3THR) , `pthread_rwlock_unlock`(3THR) , `pthread_rwlock_wrlock`(3THR) , `pthread_rwlockattr_init`(3THR) , `attributes`(5)

**NAME**

pthread_rwlockattr_init, pthread_rwlockattr_destroy – initialize or destroy read-write lock attributes object

**SYNOPSIS**

cc -mt [ *flag...* ] *file*...- lpthread [ *library...* ]

```
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

**DESCRIPTION**

The `pthread_rwlockattr_init( )` function initializes a read-write lock attributes object *attr* with the default value for all of the attributes defined by the implementation.

Results are undefined if `pthread_rwlockattr_init( )` is called specifying an already initialized read-write lock attributes object.

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) does not affect any previously initialized read-write locks.

The `pthread_rwlockattr_destroy( )` function destroys a read-write lock attributes object. The effect of subsequent use of the object is undefined until the object is re-initialized by another call to `pthread_rwlockattr_init( )` . An implementation may cause `pthread_rwlockattr_destroy( )` to set the object referenced by attr to an invalid value.

**RETURN VALUES**

If successful, the `pthread_rwlockattr_init( )` and `pthread_rwlockattr_destroy( )` functions return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The `pthread_rwlockattr_init( )` function will fail if:

`ENOMEM` Insufficient memory exists to initialize the read-write lock attributes object.

The `pthread_rwlockattr_destroy( )` function may fail if:

`EINVAL` The value specified by *attr* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_rwlock_init`(3THR) , `pthread_rwlock_rdlock`(3THR) , `pthread_rwlock_unlock`(3THR) , `pthread_rwlock_wrlock`(3THR) , `pthread_rwlockattr_getpshared`(3THR) , `attributes`(5)

**NAME**

pthread_rwlock_init, pthread_rwlock_destroy – initialize or destroy a read-write lock object

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*- lpthread [ *library...* ]

#include <pthread.h>
int **pthread_rwlock_init**(pthread_rwlock_t *rwlock,const pthread_rwlockattr_t **attr*);

int **pthread_rwlock_destroy**(pthread_rwlock_t **rwlock*);
pthread_rwlock_t *rwlock*=PTHREAD_RWLOCK_INITIALIZER;

**DESCRIPTION**

The pthread_rwlock_init( ) function initializes the read-write lock referenced by *rwlock* with the attributes referenced by *attr* . If *attr* is NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used any number of times without being re-initialized. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked. Results are undefined if pthread_rwlock_init( ) is called specifying an already initialized read-write lock. Results are undefined if a read-write lock is used without first being initialized.

If the pthread_rwlock_init( ) function fails, *rwlock* is not initialized and the contents of *rwlock* are undefined.

The pthread_rwlock_destroy( ) function destroys the read-write lock object referenced by *rwlock* and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is re-initialized by another call to pthread_rwlock_init( ) . An implementation may cause pthread_rwlock_destroy( ) to set the object referenced by *rwlock* to an invalid value. Results are undefined if pthread_rwlock_destroy( ) is called when any thread holds *rwlock* . Attempting to destroy an uninitialized read-write lock results in undefined behaviour. A destroyed read-write lock object can be re-initialized using pthread_rwlock_init( ) ; the results of otherwise referencing the read-write lock object after it has been destroyed are undefined.

In cases where default read-write lock attributes are appropriate, the macro PTHREAD_RWLOCK_INITIALIZER can be used to initialize read-write locks that are statically allocated. The effect is equivalent to dynamic initialization by a call to pthread_rwlock_init( ) with the parameter *attr* specified as NULL, except that no error checks are performed.

**RETURN VALUES**

If successful, the pthread_rwlock_init( ) and pthread_rwlock_destroy( ) functions return 0 . Otherwise, an error number is returned to indicate the error.

**ERRORS**

The pthread_rwlock_init( ) and pthread_rwlock_init( ) functions will fail if:

EINVAL The value specified by *attr* is invalid.

EINVAL The value specified by *rwlock* is invalid.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_rwlock_rdlock(3THR) , pthread_rwlock_unlock(3THR) , pthread_rwlock_wrlock(3THR) , pthread_rwlockattr_init(3THR) , attributes(5)

**NAME**

pthread_rwlock_rdlock, pthread_rwlock_tryrdlock – lock or attempt to lock a read-write lock object for reading

**SYNOPSIS**

cc -mt [ *flag*... ] *file*...- lpthread [ *library*... ]

#include <pthread.h>
int **pthread_rwlock_rdlock**(pthread_rwlock_t **rwlock*);

int **pthread_rwlock_tryrdlock**(pthread_rwlock_t **rwlock*);

**DESCRIPTION**

The `pthread_rwlock_rdlock( )` function applies a read lock to the read-write lock referenced by *rwlock* . The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the `pthread_rwlock_rdlock( )` call) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on *rwlock* at the time the call is made.

Implementations are allowed to favors writers over readers to avoid writer starvation. The current implementation favors writers over readers.

A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the `pthread_rwlock_rdlock( )` function *n* times). If so, the thread must perform matching unlocks (that is, it must call the `pthread_rwlock_unlock( )` function *n* times).

The function `pthread_rwlock_tryrdlock( )` applies a read lock as in the `pthread_rwlock_rdlock( )` function with the exception that the function fails if any thread holds a write lock on *rwlock* or there are writers blocked on *rwlock* .

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

**RETURN VALUES**

If successful, the `pthread_rwlock_rdlock( )` function returns `0` . Otherwise, an error number is returned to indicate the error.

The function `pthread_rwlock_tryrdlock( )` returns `0` if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the error.

**ERRORS**

The `pthread_rwlock_tryrdlock( )` function will fail if:

| | |
|---|---|
| EBUSY | The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_rwlock_init(3THR) , pthread_rwlock_wrlock(3THR) , pthread_rwlockattr_init(3THR) , pthread_rwlock_unlock(3THR) , attributes(5)

**NAME**

pthread_rwlock_unlock – unlock a read-write lock object

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
int **pthread_rwlock_unlock**(pthread_rwlock_t **rwlock*);

**DESCRIPTION**

The `pthread_rwlock_unlock( )` function is called to release a lock held on the read-write lock object referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

If this function is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this function releases the calling thread's last read lock on this read-write lock object, then the calling thread is no longer one of the owners of the object. If this function releases the last read lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If this function is called to release a write lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If the call to the `pthread_rwlock_unlock( )` function results in the read-write lock object becoming unlocked and there are multiple threads waiting to acquire the read-write lock object for writing, the scheduling policy is used to determine which thread acquires the read-write lock object for writing. If there are multiple threads waiting to acquire the read-write lock object for reading, the scheduling policy is used to determine the order in which the waiting threads acquire the read-write lock object for reading. If there are multiple threads blocked on *rwlock* for both read locks and write locks, it is unspecified whether the readers acquire the lock first or whether a writer acquires the lock first.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

**RETURN VALUES**

If successful, the `pthread_rwlock_unlock( )` function returns `0`. Otherwise, an error number is returned to indicate the error.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_rwlock_init`(3THR), `pthread_rwlock_rdlock`(3THR), `pthread_rwlock_wrlock`(3THR), `pthread_rwlockattr_init`(3THR), `attributes`(5)

**NAME**

pthread_rwlock_wrlock, pthread_rwlock_trywrlock – lock or attempt to lock a read-write lock object for writing

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*- lpthread [ *library...* ]

#include <pthread.h>
int **pthread_rwlock_wrlock**(pthread_rwlock_t **rwlock*);

int **pthread_rwlock_trywrlock**(pthread_rwlock_t **rwlock*);

**DESCRIPTION**

The `pthread_rwlock_wrlock()` function applies a write lock to the read-write lock referenced by *rwlock* . The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rwlock* . Otherwise, the thread blocks (that is, does not return from the `pthread_rwlock_wrlock()` call) until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

Implementations are allowed to favor writers over readers to avoid writer starvation. The current implementation favors writers over readers.

The function `pthread_rwlock_trywrlock()` applies a write lock like the `pthread_rwlock_wrlock()` function, with the exception that the function fails if any thread currently holds *rwlock* (for reading or writing).

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

**RETURN VALUES**

If successful, the `pthread_rwlock_wrlock()` function returns `0` . Otherwise, an error number is returned to indicate the error.

The function `pthread_rwlock_trywrlock()` returns `0` if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the error.

**ERRORS**

The `pthread_rwlock_trywrlock()` function will fail if:

`EBUSY` The read-write lock could not be acquired for writing because it was already locked for reading or writing.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

pthread_rwlock_init(3THR) , pthread_rwlock_unlock(3THR) , pthread_rwlockattr_init(3THR) , pthread_rwlock_rdlock(3THR) , attributes(5)

**NAME**

pthread_self – get calling thread's ID

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
pthread_t **pthread_self**(void););

**DESCRIPTION**

The `pthread_self()` function returns the thread ID of the calling thread.

**ERRORS**

No errors are defined.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`pthread_create`(3THR), `pthread_equal`(3THR), `attributes`(5), `standards`(5)

**NAME**

pthread_setcancelstate – enable or disable cancellation

**SYNOPSIS**

cc –mt [ *flag...* ] *file...*– lpthread [ *library...* ]

#include <pthread.h>
int **pthread_setcancelstate**(int*state*, int **oldstate*);

**DESCRIPTION**

pthread_setcancelstate( ) atomically sets the calling thread's cancellation state to the specified *state* and if *oldstate* is not NULL, stores the previous cancellation *state* in *oldstate*.

The *state* can be either of the following:

PTHREAD_CANCEL_ENABLE This is the default. When cancellation is deferred (deferred cancellation is also the default), cancellation occurs when the target thread reaches a cancellation point and a cancel is pending. When cancellation is asynchronous, receipt of a pthread_cancel(3THR) call causes immediate cancellation.

PTHREAD_CANCEL_DISABLE When cancellation is deferred, all cancellation requests to the target thread are held pending. When cancellation is asynchronous, all cancellation requests to the target thread are held pending; as soon as cancellation is re-enabled, pending cancellations are executed immediately.

See cancellation(3THR) for the definition of a cancellation point and a discussion of cancellation concepts. See pthread_setcanceltype(3THR) for explanations of deferred and asynchronous cancellation.

pthread_setcancelstate( ) is a cancellation point when it is called with PTHREAD_CANCEL_ENABLE and the cancellation type is PTHREAD_CANCEL_ASYNCHRONOUS.

**RETURN VALUES**

Upon successful completion, pthread_setcancelstate( ), returns 0. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The pthread_setcancelstate( ) function will fail if:

EINVAL The specified *state* is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

cancellation(3THR), condition(3THR), pthread_cancel(3THR), pthread_cleanup_pop(3THR), pthread_cleanup_push(3THR), pthread_exit(3THR), pthread_join(3THR), pthread_setcanceltype(3THR), pthread_testcancel(3THR), setjmp(3C), attributes(5)

**NAME**

pthread_setcanceltype – set the cancellation type of a thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
int **pthread_setcanceltype**(int*type*, int **oldtype*);

**DESCRIPTION**

pthread_setcanceltype( ) atomically sets the calling thread's cancellation type to the specified type and, if *oldtype* is not NULL, stores the previous cancellation type in *oldtype*. The type can be either of the following:

PTHREAD_CANCEL_DEFERRED — This is the default. When cancellation is enabled (enabled cancellation is also the default), cancellation occurs when the target thread reaches a cancellation point and a cancel is pending. When cancellation is disabled, all cancellation requests to the target thread are held pending.

PTHREAD_CANCEL_ASYNCHRONOUS — When cancellation is enabled, receipt of a pthread_cancel(3THR) call causes immediate cancellation. When cancellation is disabled, all cancellation requests to the target thread are held pending; as soon as cancellation is re-enabled, pending cancellations are executed immediately.

See cancellation(3THR) for the definition of a cancellation point and a discussion of cancellation concepts. See pthread_setcancelstate(3THR) for explanations of enabling and disabling cancellation.

pthread_setcanceltype( ) is a cancellation point if type is called with PTHREAD_CANCEL_ASYNCHRONOUS and the cancellation state is PTHREAD_CANCEL_ENABLE.

**RETURN VALUES**

Upon successful completion, the pthread_setcanceltype( ) function returns 0. Otherwise, an error number is returned to indicate the error.

**ERRORS**

The pthread_setcanceltype( ) function will fail if:

EINVAL — The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

cancellation(3THR), condition(3THR), pthread_cancel(3THR), pthread_cleanup_pop(3THR), pthread_cleanup_push(3THR), pthread_exit(3THR), pthread_join(3THR), pthread_setcancelstate(3THR), pthread_testcancel(3THR), setjmp(3C), attributes(5)

**NAME**

pthread_sigmask – change or examine calling thread's signal mask

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

**DESCRIPTION**

The `pthread_sigmask()` function changes or examines a calling thread's signal mask. Each thread has its own signal mask. A new thread inherits the calling thread's signal mask and priority; however, pending signals are not inherited. Signals pending for a new thread will be empty.

If the value of the argument `set` is not `NULL`, `set` points to a set of signals that can modify the currently blocked set. If the value of `set` is `NULL`, the value of *how* is insignificant and the thread's signal mask is unmodified; thus, `pthread_sigmask()` can be used to inquire about the currently blocked signals.

The value of the argument *how* specifies the method in which the set is changed and takes one of the following values:

`SIG_BLOCK` — `set` corresponds to a set of signals to block. They are added to the current signal mask.

`SIG_UNBLOCK` — `set` corresponds to a set of signals to unblock. These signals are deleted from the current signal mask.

`SIG_SETMASK` — `set` corresponds to the new signal mask. The current signal mask is replaced by `set`.

If the value of *oset* is not `NULL`, it points to the location where the previous signal mask is stored.

**RETURN VALUES**

Upon successful completion, the `pthread_sigmask()` function returns `0`. Otherwise, it returns a non-zero value.

**ERRORS**

The `pthread_sigmask()` function will fail if:

`EINVAL` — The value of *how* is not defined and *oset* is `NULL.`

**EXAMPLES**

**EXAMPLE 1** The following example shows how to create a default thread that can serve as a signal catcher / handler with its own signal mask. `new` will have a different value from the creator's signal mask.

As POSIX threads and Solaris threads are fully compatible even within the same process, this example uses `pthread_create`(3THR) if you execute `a.out 0`, or `thr_create`(3THR) if you execute `a.out 1`.

In this example:

- `sigemptyset`(3C) initializes a null signal set, `new`. `sigaddset`(3C) packs the signal, `SIGINT`, into that new set.
- Either `pthread_sigmask( )` or `thr_sigsetmask( )` is used to mask the signal, `SIGINT` (CTRL-C), from the calling thread, which is `main( )`. The signal is masked to guarantee that only the new thread will receive this signal.
- `pthread_create( )` or `thr_create( )` creates the signal-handling thread.
- Using `pthread_join`(3THR) or `thr_join`(3THR), `main( )` then waits for the termination of that signal-handling thread, whose ID number is `user_threadID;` after which, `main( )` will `sleep`(3C) for 2 seconds, and then the program terminates.
- The signal-handling thread, `handler`:
  - Assigns the handler `interrupt( )` to handle the signal `SIGINT`, by the call to `sigaction`(2).
  - Resets its own signal set to *not block* the signal, `SIGINT`.
  - Sleeps for 8 seconds to allow time for the user to deliver the signal, `SIGINT`, by pressing the CTRL-C.

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic first 3-lines for threads */
#include <pthread.h>
#include <thread.h>
thread_t user_threadID;
sigset_t new;
void *handler( ), interrupt( );

main( int argc, char *argv[ ] )  {
 test_argv(argv[1]);

 sigemptyset(&new);
 sigaddset(&new, SIGINT);
 switch(*argv[1])  {

  case '0':   /* POSIX */
   pthread_sigmask(SIG_BLOCK, &new, NULL);
   pthread_create(&user_threadID, NULL, handler, argv[1]);
   pthread_join(user_threadID, NULL);
   break;

  case '1':   /* Solaris */
   thr_sigsetmask(SIG_BLOCK, &new, NULL);
   thr_create(NULL, 0, handler, argv[1], 0, &user_threadID);
   thr_join(user_threadID, NULL, NULL);
   break;
  }  /* switch */

 printf("thread handler, # %d, has exited\n",user_threadID);
 sleep(2);
 printf("main thread, # %d is done\n", thr_self( ));
```

```
} /* end main */

struct sigaction act;

void *
handler(char argv1[ ])
{
 act.sa_handler = interrupt;
 sigaction(SIGINT, &act, NULL);
 switch(*argv1) {
  case '0':     /* POSIX */
   pthread_sigmask(SIG_UNBLOCK, &new, NULL);
   break;
  case '1':   /* Solaris */
   thr_sigsetmask(SIG_UNBLOCK, &new, NULL);
   break;
 }
 printf("\n Press CTRL-C to deliver SIGINT signal to the process\n");
 sleep(8);  /* give user time to hit CTRL-C */
}

void
interrupt(int sig)
{
 printf("thread %d caught signal %d\n", thr_self( ), sig);
}

void test_argv(char argv1[ ])    {
 if(argv1 == NULL)  {
  printf("use 0 as arg1 to use thr_create( );\n \
  or use 1 as arg1 to use pthread_create( )\n");
  exit(NULL);
 }
}
```

**EXAMPLE 2**

In the last example, the `handler` thread served as a signal-handler while also taking care of activity of its own (in this case, sleeping, although it could have been some other activity). A thread could be completely dedicated to signal-handling simply by waiting for the delivery of a selected signal by blocking with `sigwait`(2). The two subroutines in the previous example, `handler( )` and `interrupt( )`, could have been replaced with the following routine:

```
void *
handler( )
{ int signal;
 printf("thread %d is waiting for you to press the CTRL-C keys\n", thr_self( ));
 sigwait(&new, &signal);
 printf("thread %d has received the signal %d \n", thr_self( ), signal);
}
 /*  pthread_create( ) and thr_create( ) would use NULL instead of argv[1]
   for the arg passed to handler( ) */
```

In this routine, one thread is dedicated to catching and handling the signal specified by the set `new`, which allows `main()` and all of its other sub-threads, created *after* `pthread_sigmask()` or `thr_sigsetmask()` masked that signal, to continue uninterrupted. Any use of `sigwait`(2) should be such that all threads block the signals passed to `sigwait`(2) at all times. Only the thread that calls `sigwait()` will get the signals. The call to `sigwait`(2) takes two arguments.

For this type of background dedicated signal-handling routine, you may wish to use a Solaris daemon thread by passing the argument, `THR_DAEMON,` to `thr_create`(3THR).

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe and Async-Signal-Safe |

**SEE ALSO**

`sigaction`(2), `sigprocmask`(2), `sigwait`(2), `cond_wait`(3THR), `pthread_create`(3THR), `pthread_join`(3THR), `pthread_self`(3THR), `sigsetops`(3C), `sleep`(3C), `attributes`(5), `standards`(5)

**NOTES**

It is not possible to block signals that cannot be ignored (see `sigaction`(2)). If using the threads library, it is not possible to block the signals `SIGLWP` or `SIGCANCEL,` which are reserved by the threads library. Additionally, it is impossible to unblock the signal `SIGWAITING,` which is always blocked on all threads. This restriction is quietly enforced by the threads library.

Using `sigwait`(2) in a dedicated thread allows asynchronously generated signals to be managed synchronously; however, `sigwait`(2) should never be used to manage synchronously generated signals.

Synchronously generated signals are exceptions that are generated by a thread and are directed at the thread causing the exception. Since `sigwait()` blocks waiting for signals, the blocking thread cannot receive a synchronously generated signal.

If `sigprocmask`(2) is used in a multi-threaded program, it will be the same as if `pthread_sigmask()` has been called. POSIX leaves the semantics of the call to `sigprocmask`(2) unspecified in a multi-threaded process, so programs that care about POSIX portability should not depend on this semantic.

If a signal is delivered while a thread is waiting on a condition variable, the `cond_wait()` will be interrupted (see `cond_wait`(3THR)) and the handler will be executed. The handler should assume that the lock protecting the condition variable is held.

Although `pthread_sigmask()` is Async-Signal-Safe with respect to the Solaris environment, this safeness is not guaranteed to be portable to other POSIX domains.

Signals which are generated synchronously should not be masked. If such a signal is blocked and delivered, the receiving process is killed.

A thread directed `SIGALRM` generated because of a realtime interval timer or process alarm clock is not maskable by a signal masking function, such as thr_sigsetmask(3T), or sigprocmask(2). See `alarm(2)` and `setitimer`(2).

**NAME**

pthread_testcancel – create cancellation point in the calling thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...– lpthread [ *library*... ]

#include <pthread.h>
void **pthread_testcancel** ();

**DESCRIPTION**

The pthread_testcancel( ) function forces testing for cancellation. This is useful when you need to execute code that runs for long periods without encountering cancellation points; such as a library routine that executes long-running computations without cancellation points. This type of code can block cancellation for unacceptable long periods of time. One strategy for avoiding blocking cancellation for long periods, is to insert calls to pthread_testcancel( ) in the long-running computation code and to setup a cancellation handler in the library code, if required.

**RETURN VALUES**

The pthread_testcancel( ) function returns a void.

**ERRORS**

The pthread_testcancel( ) function does not return errors.

**EXAMPLES**

**EXAMPLE 1** See cancellation(3THR) for an example of using pthread_testcancel( ) to force testing for cancellation and a discussion of cancellation concepts.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

Intro(3), cancellation(3THR), condition(3THR), pthread_cleanup_pop(3THR), pthread_cleanup_push(3THR), pthread_exit(3THR), pthread_join(3THR), pthread_setcancelstate(3THR), pthread_setcanceltype(3THR), setjmp(3C), attributes(5)

**NOTES**

pthread_testcancel( ) has no effect if cancellation is disabled.

Use pthread_testcancel( ) with pthread_setcanceltype( ) called with its canceltype set to PTHREAD_CANCEL_DEFERRED. pthread_testcancel( ) operation is undefined if pthread_setcanceltype( ) was called with its canceltype argument set to PTHREAD_CANCEL_ASYNCHRONOUS.

It is possible to kill a thread when it is holding a resource, such as lock or allocated memory. If that thread has not setup a cancellation cleanup handler to release the held resource, the application is "cancel-unsafe". See attributes(5) for a discussion of Cancel-Safety, Deferred-Cancel-Safety, and Asynchronous-Cancel-Safety.

## NAME

rwlock, rwlock_init, rwlock_destroy, rw_rdlock, rw_wrlock, rw_tryrdlock, rw_trywrlock, rw_unlock – multiple readers, single writer locks

## SYNOPSIS

cc -mt [ *flag...* ] *file*...[ *library...* ]

```
#include <synch.h>
int rwlock_init(rwlock_t *rwlp, int type, void * arg);

int rwlock_destroy(rwlock_t *rwlp);

int rw_rdlock(rwlock_t *rwlp);

int rw_wrlock(rwlock_t *rwlp);

int rw_unlock(rwlock_t *rwlp);

int rw_tryrdlock(rwlock_t *rwlp);

int rw_trywrlock(rwlock_t *rwlp);
```

## DESCRIPTION

Many threads can have simultaneous read-only access to data, while only one thread can have write access at any given time. Multiple read access with single write access is controlled by locks, which are generally used to protect data that is frequently searched.

Readers / writer locks can synchronize threads in this process and other processes if they are allocated in writable memory and shared among cooperating processes (see `mmap`(2) ), and are initialized for this purpose.

Additionally, readers / writer locks must be initialized prior to use. `rwlock_init()` The readers / writer lock pointed to by *rwlp* is initialized by `rwlock_init()` . A readers / writer lock is capable of having several types of behavior, which is specified by `type` . *arg* is currently not used, although a future type may define new behavior parameters by way of *arg* .

`type` may be one of the following:

`USYNC_PROCESS`
: The readers / writer lock can synchronize threads in this process and other processes. The readers / writer lock should be initialized by only one process. *arg* is ignored. A readers / writer lock initialized with this type, must be allocated in memory shared between processses, i.e. either in Sys V shared memory (see `shmop`(2) ) or in memory mapped to a file (see `mmap`(2) ). It is illegal to initialize the object this way and to not allocate it in such shared memory.

`USYNC_THREAD`
: The readers / writer lock can synchronize threads in this process, only. *arg* is ignored.

Additionally, readers / writer locks can be initialized by allocation in zeroed memory. A `type` of `USYNC_THREAD` is assumed in this case. Multiple threads must not simultaneously initialize the same readers / writer lock. And a readers / writer lock must not be re-initialized while in use by other threads.

```
The following are default readers/writer lock initialization (intra-process):
   rwlock_t rwlp;
   rwlock_init(&rwlp, NULL, NULL);
    OR
   rwlock_init(&rwlp, USYNC_THREAD, NULL);
    OR
   rwlock_t  rwlp  =  DEFAULTRWLOCK;

The following is a customized readers/writer lock
initialization (inter-process):
```

`rwlock_init(&rwlp, USYNC_PROCESS , NULL );`

Any state associated with the readers / writer lock pointed to by *rwlp* are destroyed by `rwlock_destroy( )` and the readers / writer lock storage space is not released.

`rw_rdlock( )` gets a read lock on the readers / writer lock pointed to by *rwlp* . If the readers / writer lock is currently locked for writing, the calling thread blocks until the write lock is freed. Multiple threads may simultaneously hold a read lock on a readers / writer lock.

`rw_tryrdlock( )` trys to get a read lock on the readers / writer lock pointed to by *rwlp* . If the readers / writer lock is locked for writing, it returns an error; otherwise, the read lock is acquired.

`rw_wrlock( )` gets a write lock on the readers / writer lock pointed to by *rwlp* . If the readers / writer lock is currently locked for reading or writing, the calling thread blocks until all the read and write locks are freed. At any given time, only one thread may have a write lock on a readers / writer lock.

`rw_trywrlock( )` trys to get a write lock on the readers / writer lock pointed to by *rwlp* . If the readers / writer lock is currently locked for reading or writing, it returns an error.

`rw_unlock( )` unlocks a readers / writer lock pointed to by *rwlp* , if the readers / writer lock is locked and the calling thread holds the lock for either reading or writing. One of the other threads that is waiting for the readers / writer lock to be freed will be unblocked, provided there is other waiting threads. If the calling thread does not hold the lock for either reading or writing, no error status is returned, and the program's behavior is unknown.

**RETURN VALUES**

If successful, these functions return `0` . Otherwise, a non-zero value is returned to indicate the error.

**ERRORS**

The `rwlock_init( )` function will fail if:

`EINVAL` `type` is invalid.

The `rw_tryrdlock( )` or `rw_trywrlock( )` functions will fail if:

`EBUSY` The reader or writer lock pointed to by *rwlp* was already locked.

These functions may fail if:

`EFAULT` *rwlp* or *arg* points to an illegal address.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`mmap`(2) , `attributes`(5)

**NOTES**

These interfaces also available by way of:

`#include <thread.h>`

If multiple threads are waiting for a readers / writer lock, the acquisition order is random by default. However, some implementations may bias acquisition order to avoid depriving writers. The current implementation favors writers over readers.

**NAME**

schedctl_init, schedctl_lookup, schedctl_exit, schedctl_start, schedctl_stop – preemption control

**SYNOPSIS**

`cc` [ *flag* ... ] *file* ... `–lsched` [ *library* ... ]
#include <schedctl.h>
schedctl_t ***schedctl_init**(void);

schedctl_t ***schedctl_lookup**(void);

void **schedctl_exit**(void);

void **schedctl_start**(schedctl_t **ptr*);

void **schedctl_stop**(schedctl_t **ptr*);

**DESCRIPTION**

These functions provide limited control over the scheduling of a *lightweight process* (LWP). They allow a running LWP to give a hint to the kernel that preemptions of that LWP should be avoided. The most likely use for these functions is to block preemption while holding a spinlock. Improper use of this facility, including attempts to block preemption for sustained periods of time, may result in reduced performance.

`schedctl_init()` initializes preemption control for the calling LWP and returns a pointer used to refer to the data. If `schedctl_init()` is called more than once by the same LWP, the most recently returned pointer is the only valid one.

`schedctl_lookup()` returns the currently allocated preemption control data associated with the calling LWP that was previously returned by `schedctl_init()`. This can be useful in programs where it is difficult to maintain local state for each LWP.

`schedctl_exit()` removes the preemption control data associated with the calling LWP.

`schedctl_start()` is a macro that gives a hint to the kernel scheduler that preemption should be avoided on the current LWP. The pointer passed to the macro must be the same as the pointer returned by the call to `schedctl_init()` by the current LWP. The behavior of the program when other values are passed is undefined.

`schedctl_stop()` is a macro that removes the hint that was set by `schedctl_start()`. As with `schedctl_start()`, the pointer passed to the macro must be the same as the pointer returned by the call to `schedctl_init()` by the current LWP.

`schedctl_start()` and `schedctl_stop()` are intended to be used to bracket short critical sections, such as the time spent holding a spinlock. Other uses, including the failure to call `schedctl_stop( )` soon after calling `schedctl_start( )` , may result in poor performance.

**RETURN VALUES**

`schedctl_init( )` returns a pointer to a `schedctl_t` structure if the initialization was successful, or `NULL` otherwise. `schedctl_lookup( )` returns a pointer to a `schedctl_t` structure if the data for that LWP was found, or `NULL` otherwise.

**ERRORS**

None returned.

**SEE ALSO**

`priocntl`(1) , `exec`(2) , `fork`(2) , `priocntl`(2) , `thr_create`(3THR)

**NOTES**

Preemption control is intended for use by LWPs belonging to the time-sharing (TS) and interactive (IA) scheduling classes. If used by LWPs in other scheduling classes, such as real-time (RT), no errors will be returned but `schedctl_start( )` and `schedctl_stop( )` will not have any effect.

Use of preemption control by unbound threads in multithreaded applications (see `thr_create`(3THR) ) is not supported and will result in undefined behavior.

The data used for preemption control is not copied in the child of a `fork`(2) . Thus, if a process containing LWPs using preemption control calls `fork` , and the child does not immediately call `exec`(2) , each LWP in the child must call `schedctl_init( )` again prior to any future uses of `schedctl_start( )` and `schedctl_stop( )` . Failure to do so will result in undefined behavior.

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

sched_getscheduler(3RT), sched_setparam(3RT), sched_setscheduler(3RT), attributes(5), sched(3HEAD)

**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

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 |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`sched_getparam`(3RT) , `sched_setparam`(3RT) , `sched_getscheduler`(3RT) , `sched_rr_get_interval`(3RT) , `sched_setscheduler`(3RT) , `attributes`(5) , `sched`(3HEAD) , `time`(3HEAD)

## 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**

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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

sched_getparam(3RT), sched_setparam(3RT), sched_setscheduler(3RT), attributes(5), sched(3HEAD)

**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**

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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

sched_getparam(3RT), sched_setparam(3RT), sched_get_priority_max(3RT), sched_getscheduler(3RT), sched_setscheduler(3RT), attributes(5), sched(3HEAD)

**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**

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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`intro`(2), `exec`(2), `sched_getparam`(3RT), `sched_getscheduler`(3RT), `sched_setscheduler`(3RT), `attributes`(5), `sched`(3HEAD)

**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**

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*. 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 file `<sched.h>`:

`SCHED_FIFO` (realtime), First-In-First-Out; processes scheduled to this policy, if not pre-empted by a higher priority or interrupted by a signal, will proceed until completion.

`SCHED_RR` (realtime), Round-Robin; processes scheduled to this policy, if not pre-empted by a higher priority or interrupted by a signal, will execute for a time period, returned by `sched_rr_get_interval`(3RT) or by the system.

`SCHED_OTHER` (time-sharing)

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 system supports a two-level scheduling model in which library threads are multiplexed on top of several kernel scheduled entities. 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 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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

priocntl(1), intro(2), exec(2), priocntl(2), sched_get_priority_max(3RT), sched_getparam(3RT), sched_getscheduler(3RT), sched_setparam(3RT), attributes(5), sched(3HEAD)

**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**

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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

attributes(5), sched(3HEAD)

**NAME**

semaphore, sema_init, sema_destroy, sema_wait, sema_trywait, sema_post – semaphores

**SYNOPSIS**

cc [ *flag*... ] *file*...- lthread - lc [ *library*... ]

#include <synch.h>
int **sema_init**(sema_t **sp*, unsigned int *count*, int *type*, void * *arg*);

int **sema_destroy**(sema_t **sp*);

int **sema_wait**(sema_t **sp*);

int **sema_trywait**(sema_t **sp*);

int **sema_post**(sema_t **sp*);

**DESCRIPTION**

A semaphore is a non-negative integer count and is generally used to coordinate access to resources. The initial semaphore count is set to the number of free resources, then threads slowly increment and decrement the count as resources are added and removed. If the semaphore count drops to zero, which means no available resources, threads attempting to decrement the semaphore will block until the count is greater than zero.

Semaphores can synchronize threads in this process and other processes if they are allocated in writable memory and shared among the cooperating processes (see `mmap`(2) ), and have been initialized for this purpose.

Semaphores must be initialized before use; semaphores pointed to by *sp* to *count* are initialized by `sema_init()` . The *type* argument can assign several different types of behavior to a semaphore. No current type uses *arg* , although it may be used in the future.

The *type* argument may be one of the following:

`USYNC_PROCESS` The semaphore can synchronize threads in this process and other processes. Initializing the semaphore should be done by only one process. A semaphore initialized with this type must be allocated in memory shared between processes, i.e. either in Sys V shard memory (see `shmop`(2) ), or in memory mapped to a file (see `mmap`(2) ). It is illegal to initialize the object this way and to not allocate it in such shared memory. *arg* is ignored.

`USYNC_THREAD` The semaphore can synchronize threads only in this process. The *arg* argument is ignored. `USYNC_THREAD` does not support multiple mappings to the same logical synch object. If you need to `mmap()` a synch object to different locations within the same address space, then the synch object should be initialized as a shared object `USYNC_PROCESS` for Solaris threads and `PTHREAD_PROCESS_PRIVATE` for POSIX threads.

A semaphore must not be simultaneously initialized by multiple threads, nor re-initialized while in use by other threads.

Default semaphore initialization (intra-process):

```
sema_t sp;
int count  =  1;
sema_init(&sp, count, NULL, NULL);
```

or

```
sema_init(&sp, count, USYNC_THREAD, NULL);
```

Customized semaphore initialization (inter-process):

```
sema_t sp;
int count  =  1;
sema_init(&sp, count, USYNC_PROCESS, NULL);
```

The `sema_destroy()` function destroys any state related to the semaphore pointed to by *sp* . The semaphore storage space is not released.

The `sema_wait()` function blocks the calling thread until the semaphore count pointed to by *sp* is greater than zero, and then it atomically decrements the count.

The `sema_trywait()` function atomically decrements the semaphore count pointed to by *sp* , if the count is greater than zero; otherwise, it returns an error.

The `sema_post()` function atomically increments the semaphore count pointed to by *sp* . If there are any threads blocked on the semaphore, one will be unblocked.

The semaphore functionality described on this man page is for the Solaris threads implementation. For the POSIX-compliant semaphore interface documentation, see `sem_open`(3RT) , `sem_init`(3RT) , `sem_wait`(3RT) , `sem_post`(3RT) , `sem_getvalue`(3RT) , `sem_unlink`(3RT) , `sem_close`(3RT) , `sem_destroy`(3RT) ).

**RETURN VALUES**

Upon successful completion, `0` is returned; otherwise, a non-zero value indicates an error.

**ERRORS**

These functions will fail if:

EINVAL The *sp* argument does not refer to a valid semaphore..

EFAULT Either the *sp* or *arg* argument points to an illegal address.

The `sema_wait()` function will fail if::

EINTR The wait was interrupted by a signal or `fork()` .

The `sema_trywait()` function will fail if::

EBUSY The semaphore pointed to by *sp* has a zero count.

The `sema_post()` function will fail if:

EOVERFLOW The semaphore value pointed to by *sp* exceeds `SEM_VALUE_MAX` .

## EXAMPLES

**EXAMPLE 1** The customer waiting-line in a bank is analogous to the synchronization scheme of a semaphore using `sema_wait()` and `sema_trywait()` :

```
/* cc [ flag ... ] file ... -lthread [ library ... ] */
#include <errno.h>
#define TELLERS 10
sema_t     tellers;     /* semaphore */
int banking_hours(), deposit_withdrawal;
void*customer(), do_business(), skip_banking_today();
...

sema_init(&tellers, TELLERS, USYNC_THREAD, NULL);
    /* 10 tellers available */
while(banking_hours())
    pthread_create(NULL, NULL, customer, deposit_withdrawal);
...

void *
customer(int deposit_withdrawal)
{
     int this_customer, in_a_hurry = 50;
     this_customer = rand() % 100;

     if (this_customer == in_a_hurry)  {
             if (sema_trywait(&tellers) != 0)
                   if (errno == EAGAIN){ /* no teller available */
                                skip_banking_today(this_customer);
                                return;
              } /* else go immediately to available teller and
                                decrement tellers */
      }
      else
             sema_wait(&tellers); /* wait for next teller, then proceed,
                                       and decrement tellers */
```

```
        do_business(deposit_withdrawal);
        sema_post(&tellers); /* increment tellers;
                                this_customer's teller
                                is now available */
  }
```

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Async-Signal-Safe |

**SEE ALSO**

mmap(2) , shmop(2) , sem_close(3RT) , sem_destroy(3RT) , sem_getvalue(3RT) , sem_init(3RT) , sem_open(3RT) , sem_post(3RT) , sem_unlink(3RT) , sem_wait(3RT) , attributes(5) , standards(5)

**NOTES**

These functions are also available by way of:

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

By default, there is no defined order of unblocking for multiple threads waiting for a semaphore.

**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 |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO**

sem_init(3RT), sem_open(3RT), sem_unlink(3RT), attributes(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.

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

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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

sem_init(3RT), sem_open(3RT), attributes(5)

**NAME**

sem_getvalue – get the value of a semaphore

**SYNOPSIS**

cc [ *flag*... ] *file*... –lrt [ *library*... ]
#include <semaphore.h>
int **sem_getvalue**(sem_t **sem*, int **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 processed 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 |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO**

sem_post(3RT), sem_wait(3RT), attributes(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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

sem_destroy(3RT), sem_post(3RT), sem_wait(3RT), attributes(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 |
|---|---|
| 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)

**NOTES**

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

**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`(3THR)).

**EXAMPLES**

**EXAMPLE 1** See `sem_wait`(3RT).

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Async-Signal-Safe |

**SEE ALSO**

sched_setscheduler(3RT), sem_wait(3RT), semaphore(3THR), 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_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 |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

exec(2), exit(2), sem_close(3RT), sem_open(3RT), attributes(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()`:

/* `cc` [ *flag* ... ] `file` ... `-lrt` `-lthread` [ *library* ... ] */

```
#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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO** sem_post(3RT) , attributes(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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

close(2), dup(2), exec(2), fcntl(2), mmap(2), umask(2), shm_unlink(3RT), sysconf(3C), attributes(5), fcntl(3HEAD)

**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 |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

close(2), mmap(2), mlock(3C), shm_open(3RT), attributes(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 |
|---|---|
| MT-Level | Async-Signal-Safe |

**SEE ALSO**

`kill`(2), `sigwaitinfo`(3RT), `attributes`(5), `siginfo`(3HEAD), `signal`(3HEAD)

**NAME**

sigwaitinfo, sigtimedwait – wait for queued signals

**SYNOPSIS**

`cc` [ *flag...* ] *file...* `–lrt` [ *library...* ]
#include <signal.h>
int **sigwaitinfo**(const sigset_t **set*, siginfo_t **info*);

int **sigtimedwait**(const sigset_t **set*, siginfo_t **info*, const struct timespec **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:

`ENOSYS` The functions `sigwaitinfo()` and `sigtimedwait()` are not supported by this implementation.

The `sigtimedwait()` function will also fail if:

`EAGAIN` No signal specified by `set` was generated within the specified timeout period.

The `sigwaitinfo()` and `sigtimedwait()` functions may fail if:

`EINTR` The wait was interrupted by an unblocked, caught signal. It will be documented in system documentation whether this error will cause these functions to fail.

The `sigtimedwait()` function may also 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 |
|---|---|
| MT-Level | Async-Safe |

**SEE ALSO**

`time`(2) , `sigqueue`(3RT) , `attributes`(5) , `siginfo`(3HEAD) , `signal`(3HEAD) , `time`(3HEAD)

**NAME**

td_init – performs initialization for libthread_db library of interfaces

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

```
#include <proc_service.h>
#include <thread_db.h>

td_err_e td_init();
```

**DESCRIPTION**

`td_init()` is the global initialization function for the `libthread_db()` library of interfaces. It must be called exactly once by any process using the `libthread_db()` library before any other libthread_db function can be called.

**RETURN VALUES**

`TD_OK` The `libthread_db()` library of interfaces successfully initialized.

`TD_ERR` Initialization failed.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT Level | Safe |

**SEE ALSO**

`libthread_db`(3THR), `libthread_db`(3LIB), `attributes`(5)

**NAME**

td_log – placeholder for future logging functionality

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>

void **td_log**();

**DESCRIPTION**

This function presently does nothing; it is merely a placeholder for future logging functionality in libthread_db(3THR).

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT Level | Safe |

**SEE ALSO**

libthread(3THR), libthread_db(3THR), libthread_db(3LIB), attributes(5)

**NAME**

td_sync_get_info, td_sync_setstate, td_sync_waiters – operations on a synchronization object in libthread_db

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>
td_err_e **td_sync_get_info**(const td_synchandle_t **sh_p*, td_syncinfo_t **si_p*);

td_err_e **td_sync_setstate**(const td_synchandle_t **sh_p*);

td_err_e**td_sync_waiters**(const td_synchandle_t **sh_p* ,td_thr_iter_f **cb* ,void **cb_data_p*);

**DESCRIPTION**

Synchronization objects include mutexes, condition variables, semaphores, and reader-writer locks. In the same way that thread operations use a thread handle of type `td_thrhandle_t` , operations on synchronization objects use a synchronization object handle of type `td_synchandle_t` .

The controlling process obtains synchronization object handles either by calling the function `td_ta_sync_iter( )` to obtain handles for all synchronization objects of the target process that are known to the `libthread_db` library of interfaces, or by mapping the address of a synchronization object in the address space of the target process to a handle by calling `td_ta_map_addr2sync` ().

Note that not all synchronization objects that a process uses may be known to the `libthread_db` library and returned by `td_ta_sync_iter` . A synchronization object is known to `libthread_db` only if it was ever waited on after `libthread_db` was attached to the process. For example, a mutex may have been widely used, but if no thread ever blocked waiting to acquire it, it will not be known to `libthread_db` interfaces.

The `td_sync_get_info( )` function fills in the `td_syncinfo_t` structure * *si_p* with values for the synchronization object identified by `sh_p` . The `td_syncinfo_t` structure contains the following fields:

| | |
|---|---|
| `td_thragent_t *` *si_ta_p* | The internal process handle identifying the target process through which this synchronization object handle was obtained. Synchronization objects may be process-private or process-shared. In the latter case, the same synchronization object may have multiple handles, one for each target process's "view" of the synchronization object. |

| | |
|---|---|
| `psaddr_t si_sv_addr` | The address of the synchronization object in this target process's address space. |
| `td_sync_type_e si_type` | The type of the synchronization variable: mutex, condition variable, semaphore, or reader-writer lock. |
| `int si_shared_type` | `USYNC_THREAD` if this synchronization object is process-private; `USYNC_PROCESS` if it is process-shared. |
| `td_sync_flags_t si_flags` | Flags dependent on the type of the synchronization object. |
| `int si_state.sema_count` | Semaphores only. The current value of the semaphore |
| `int si_state.nreaders` | Reader-writer locks only. The number of readers currently holding the lock, or `-1` , if a writer is currently holding the lock. |
| `int si_state.mutex_locked` | For mutexes only. Non-zero if and only if the mutex is currently locked. |
| `int si_size` | The size of the synchronization object. |
| `uchar_t si_has_waiters` | Non-zero if and only if at least one thread is blocked on this synchronization object. |

| | |
|---|---|
| `uchar_t si_is_wlocked` | For reader-writer locks only. The value is non-zero if and only if this lock is held by a writer. |
| `td_thrhandle_t si_owner` | Mutexes and reader-writer locks only. This is the thread holding the mutex, or the write lock, if this is a reader-writer lock. The value is `NULL` if no one holds the mutex or write-lock. |
| `psaddr_t si_data` | A pointer to optional data associated with the synchronization object. Currently useful only for debugging `libthread()` interfaces. |

`td_sync_setstate` modifies the state of synchronization object *si_p* , depending on the synchronization object type. For mutexes, `td_sync_setstate` is unlocked if the value is `0` . Otherwise it is locked. For semaphores, the semaphore's count is set to the value. For reader-writer locks, the reader count set to the value if value is `>0` . The count is set to write-locked if value is `-1` . It is set to unlocked if the value is `0` . Setting the state of a synchronization object from a `libthread_db` interface may cause the synchronization object's semantics to be violated from the point of view of the threads in the target process. For example, if a thread holds a mutex, and `td_sync_setstate` is used to set the mutex to unlocked, then a different thread will also be able to subsequently acquire the same mutex.

`td_sync_waiters` iterates over the set of thread handles of threads blocked on `sh_p` . The callback function *cb* is called once for each such thread handle, and is passed the thread handle and *cb_data_p* . If the callback function returns a non-zero value, iteration is terminated early. See also `td_ta_thr_iter`(3THR) .

**RETURN VALUES**

| | |
|---|---|
| `TD_OK` | The call returned successfully. |
| `TD_BADTH` | An invalid thread handle was passed in. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |

TD_ERR A libthread_db-internal error occurred.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT Level | Safe |

**SEE ALSO**

libthread_db(3THR) , td_ta_map_addr2sync(3THR) , td_ta_sync_iter(3THR) , td_ta_thr_iter(3THR) , libthread_db(3LIB) , attributes(5)

**NAME**

td_ta_enable_stats, td_ta_reset_stats, td_ta_get_stats – collect target process statistics for libthread_db

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>
td_err_e **td_ta_enable_stats**(const td_thragent_t **ta_p*, int *on_off*);

td_err_e _stats**td_ta_reset**(const td_thragent_t **ta_p*);

td_err_e **td_ta_get_stats**(const td_thragent_t **ta_p*, td_ta_stats_t **tstats*);

**DESCRIPTION**

The controlling process may request the collection of certain statistics about a target process. Statistics gathering is disabled by default; however, each target process has a `td_ta_stats_t` structure that contains up to date values when statistic gathering is enabled. `td_ta_enable_stats( )` turns statistics gathering on or off for the process identified by *ta_p* depending on whether or not `on_off` is non-zero. When statistics gathering is turned on, all statistics are implicitly reset as though `td_ta_reset_stats( )` had been called. Statistics are not reset when statistics gathering is turned off. Except for `nthreads and r_concurrency` , the values do not change further, but they remain available for inspection by way of `td_ta_get_stats( )` . `td_ta_reset_stats( )` resets all counters in the `td_ta_stats_t` structure to zero for the target process. `td_ta_get_stats( )` returns the `td_ta_stats_t` structure for the process in * stats_t . The `td_ta_stats_t` structure is defined as follows:

```
typedef struct {
   int nthreads;         /* total number of threads in use */
   int r_concurrency;     /* requested concurrency level */
   int nrunnable_num;     /* numerator of avg. runnable threads */
   int nrunnable_den;     /* denominator of avg. runnable threads */
   int a_concurrency_num; /* numerator, avg. achieved concurrency */
   int a_concurrency_den;  /* denominator, avg. achieved concurrency */
   int nlwps_num;        /* numerator, average number of LWPs in use */
   int nlwps_den;          /* denominator, avg. number of LWPs in use */
   int nidle_num;          /* numerator, avg. number of idling LWPs */
   int nidle_den;         /* denominator, avg. number of idling LWPs */
} td_ta_stats_t;
```

*nthreads* is the number of threads that are currently part of the target process. *r_concurrency* is the current requested concurrency level, such as would be returned by `thr_setconcurrency`(3THR) The remaining fields are averages over time, each expressed as a fraction with an integral numerator and denominator. *nrunnable* is the average number of runnable threads. *a_concurrency* is the average achieved concurrency, the number of actually running threads. *a_concurrency* is less than or equal to *nrunnable* . *nlwps* is the average number of lightweight processes ( LWP s) participating in this process. It must be greater than or equal to *a_concurrency* , as every running thread is assigned to an LWP , but there may at times be additional idling LWP s with no thread assigned to them. *nidle* is the average number of idle LWP s.

**RETURN VALUES**

`TD_OK` The call completed successfully.

`TD_BADTA` An invalid internal process handle was passed in.

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_ERR` Something else went wrong.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT Level | Safe |

**SEE ALSO**

`libthread_db`(3THR) , `thr_getconcurrency`(3THR) , `libthread_db`(3LIB) , `attributes`(5)

**NAME**

td_ta_event_addr, td_thr_event_enable, td_ta_set_event, td_thr_set_event, td_ta_clear_event, td_thr_clear_event, td_ta_event_getmsg, td_thr_event_getmsg, td_event_emptyset, td_event_fillset, td_event_addset, td_event_delset, td_eventismember, td_eventisempty – thread events in libthread_db

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>
td_err_e **td_ta_event_addr**(const td_thragent_t **ta_p*, u_long event,td_notify_t **notify_p*);

td_err_e **td_thr_event_enable**(const td_thrhandle_t **th_p*, int *on_off*);

td_err_e **td_thr_set_event**(const td_thrhandle_t **th_p*, td_thr_events_t **events*);

td_err_e **td_ta_set_event**(const td_thragent_t **ta_p*, td_thr_events_t **events*);

td_err_e **td_thr_clear_event**(const td_thrhandle_t **th_p*, td_thr_events_t **events*);

td_err_e **td_ta_clear_event**(const td_thragent_t **ta_p*, td_thr_events_t **events*);

td_err_e **td_thr_event_getmsg**(const td_thrhandle_t **th_p*, td_event_msg_t **msg*);

td_err_e **td_ta_event_getmsg**(const td_thragent_t **ta_p*, td_event_msg_t **msg*);

void **td_event_emptyset**(td_thr_events_t *);

void **td_event_fillset**(td_thr_events_t *);

void **td_event_addset**(td_thr_events_t *, td_thr_events_e *n*);

void **td_event_delset**(td_thr_events_t *, td_thr_events_e *n*);

void **td_eventismember**(td_thr_events_t *, td_thr_events_e *n*);

void **td_eventisempty**(td_thr_events_t*);

**DESCRIPTION**

These routines comprise the thread event facility for `libthread_db`(3THR) . This facility allows the controlling process to be notified when certain thread-related events occur in a target process and to retrieve information associated with these events. An event consists of an event type, and optionally, some associated event data, depending on the event type. See the section titled "Event Set Manipulation Macros" that follows.

The event type and the associated event data, if any, constitute an "event message." "Reporting an event" means delivering an event message to the controlling process by way of `libthread_db` .

Several flags can control event reporting, both a per-thread and per event basis. Event reporting may further be enabled or disabled for a thread. There is not only a per-thread event mask that specifies which event types should be reported for that thread, but there is also a global event mask that applies to all threads.

An event is reported, if and only if, the executing thread has event reporting enabled, and either the event type is enabled in the executing thread's event mask, or the event type is enabled in the global event mask.

Each thread has associated with it an event buffer in which it stores the most recent event message it has generated, the type of the most recent event that it reported, and, depending on the event type, some additional information related to that event. See the section titled "Event Set Manipulation Macros" for a description of the `td_thr_events_e` and `td_event_msg_t` types and a list of the event types and the values reported with them. The thread handle, type `td_thrhandle_t` , the event type, and the possible value, together constitute an event message. Each thread's event buffer holds at most one event message.

Each event type has an event reporting address associated with it. A thread reports an event by writing the event message into the thread's event buffer and having control reach the event reporting address for that event type.

Typically, the controlling process sets a breakpoint at the event reporting address for one or more event types. When the breakpoint is hit, the controlling process knows that an event of the corresponding type has occurred.

The event types, and the additional information, if any, reported with each event, are:

| | |
|---|---|
| `TD_READY` | The thread became ready to execute. |
| `TD_SLEEP` | The thread has blocked on a synchronization object. |
| `TD_SWITCHTO` | A runnable thread is being assigned to LWP. |
| `TD_SWITCHFROM` | A running thread is being removed from its LWP. |
| `TD_LOCK_TRY` | A thread is trying to get an unavailable lock. |
| `TD_CATCHSIG` | A signal was posted to a thread. |
| `TD_IDLE` | An LWP is becoming idle. |
| `TD_CREATE` | A thread is being created. |

| | |
|---|---|
| `TD_DEATH` | A thread has terminated. |
| `TD_PREEMPT` | A thread is being preempted. |
| `TD_PRI_INHERIT` | A thread is inheriting an elevated priority from another thread. |
| `TD_REAP` | A thread is being reaped. |
| `TD_CONCURRENCY` | The number of LWPs is changing. |
| `TD_TIMEOUT` | A condition-variable timed wait expired. |

`td_ta_event_addr()` returns in * *notify_p* the event reporting address associated with event type `event` . The controlling process may then set a breakpoint at that address. If a thread hits that breakpoint, it reports an event of type `event` .

`td_thr_event_enable()` enables or disables event reporting for thread *th_p* . If a thread has event reporting disabled, it will not report any events. Threads are started with event reporting disabled. Event reporting is enabled if `on_off` is non-zero; otherwise, it is disabled. To find out whether or not event reporting is enabled on a thread, call `td_thr_getinfo()` for the thread and examine the `ti_traceme` field of the `td_thrinfo_t` structure it returns.

`td_thr_set_event()` and `td_thr_clear_event()` set and clear, respectively, a set of event types in the event mask associated with the thread *th_p* . To inspect a thread's event mask, call `td_thr_getinfo()` for the thread, and examine the `ti_events` field of the `td_thrinfo_t` structure it returns.

`td_ta_set_event()` and `td_ta_clear_event()` are just like `td_thr_set_event ()` and `td_thr_clear_event()` , respectively, except that the target process's global event mask is modified. There is no provision for inspecting the value of a target process's global event mask.

`td_thr_event_getmsg()` returns in * *msg* the event message associated with thread * *th_p* Reading a thread's event message consumes the message, emptying the thread's event buffer. As noted above, each thread's event buffer holds at most one event message; if a thread reports a second event before the first event message has been read, the second event message overwrites the first.

`td_ta_event_getmsg()` is just like `td_thr_event_getmsg()` , except that it is passed a process handle rather than a thread handle. It selects some thread that has an event message buffered, and it returns that thread's message. The thread selected is undefined, except that as long as at least one thread has an event message buffered, it will return an event message from some such thread.

**Event Set Manipulation Macros**

Several macros are provided for manipulating event sets of type `td_thr_events_t`:

| | |
|---|---|
| `td_event_emptyset` | Sets its argument to the `NULL` event set. |
| `td_event_fillset` | Sets its argument to the set of all events. |
| `td_event_addset` | Adds a specific event type to an event set. |
| `td_event_delset` | Deletes a specific event type from an event set. |
| `td_eventismember` | Tests whether a specific event type is a member of an event set. |
| `td_eventisempty` | Tests whether an event set is the `NULL` set. |

## RETURN VALUES

The following values may be returned for all thread event routines:

| | |
|---|---|
| `TD_OK` | The call returned successfully. |
| `TD_BADTH` | An invalid thread handle was passed in. |
| `TD_BADTA` | An invalid internal process handle was passed in. |
| `TD_BADPH` | There is a `NULL` external process handle associated with this internal process handle. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |
| `TD_NOMSG` | No event message was available to return to `td_thr_event_getmsg()` or `td_ta_event_getmsg()`. |
| `TD_ERR` | Some other parameter error occurred, or a `libthread_db()` internal error occurred. |

The following value may be returned for `td_thr_event_enable()`, `td_thr_set_event()`, and `td_thr_clear_event()` only:

| | |
|---|---|
| `TD_NOCAPAB` | The agent thread in the target process has not completed initialization, so this operation cannot be performed. The operation can be performed after the target process has |

been allowed to make some forward progress. See also `libthread_db`(3THR) .

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

`libthread_db`(3THR) , `libthread_db`(3LIB) , `attributes`(5)

## NAME

td_ta_get_nthreads – gets the total number of threads in a process for libthread_db

## SYNOPSIS

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

```
#include <proc_service.h>
#include <thread_db.h>
td_err_e td_ta_get_nthreads(const td_thragent_t *ta_p, int *nthread_p);
```

## DESCRIPTION

`td_ta_get_nthreads()` returns the total number of threads in process *ta_p*, including any system threads. System threads are those created by `libthread()` or `libthread_db()` on its own behalf. The number of threads is written into **nthread_p*.

## RETURN VALUES

| | |
|---|---|
| `TD_OK` | The call completed successfully. |
| `TD_BADTA` | An invalid internal process handle was passed in. |
| `TD_BADPH` | There is a `NULL` external process handle associated with this internal process handle. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |
| `TD_ERR` | *nthread_p* was `NULL,` or a `libthread_db` internal error occurred. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

## SEE ALSO

`libthread`(3THR), `libthread_db`(3THR), `libthread_db`(3LIB), `attributes`(5)

**NAME**

td_ta_map_addr2sync – get a synchronization object handle from a synchronization object's address

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>
**td_ta_map_addr2sync**(const td_thragent_t **ta_p*, psaddr_t addr,td_synchandle_t **sh_p*);

**DESCRIPTION**

`td_ta_map_addr2sync()` produces the synchronization object handle of type `td_synchandle_t` that corresponds to the address of the synchronization object (mutex, semaphore, condition variable, or reader / writer lock). Some effort is made to validate `addr` and verify that it does indeed point at a synchronization object. The handle is returned in **sh_p*.

**RETURN VALUES**

| | |
|---|---|
| `TD_OK` | The call completed successfully. |
| `TD_BADTA` | An invalid internal process handle was passed in. |
| `TD_BADPH` | There is a `NULL` external process handle associated with this internal process handle. |
| `TD_BADSH` | *sh_p* is `NULL,` or `addr` does not appear to point to a valid synchronization object. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |
| `TD_ERR` | `addr` is `NULL,` or a `libthread_db` internal error occurred. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

`libthread_db`(3THR), `libthread_db`(3LIB), `attributes`(5)

**NAME**

td_ta_map_id2thr, td_ta_map_lwp2thr – convert a thread id or LWP id to a thread handle

**SYNOPSIS**

cc [ *flag* … ] *file* … -lthread_db [ *library* … ]

#include <proc_service.h>
#include <thread_db.h>
**td_ta_map_id2thr**(const td_thragent_t **ta_p*, thread_t tid,td_thrhandle_t **th_p*);

**td_ta_map_lwp2thr**(const td_thragent_t **ta_p*, lwpid_t lwpid,td_thrhandle_t **th_p*);

**DESCRIPTION**

`td_ta_map_id2thr( )` produces the `td_thrhandle_t` thread handle that corresponds to a particular thread id, as returned by `thr_create`(3THR) or `thr_self`(3THR) . The thread handle is returned in * *th_p* .

`td_ta_map_lwp2thr( )` produces the `td_thrhandle_t` thread handle for the thread that is currently executing on the light weight process ( LWP ) and has an id of `lwpid` .

**RETURN VALUES**

| | |
|---|---|
| TD_OK | The call completed successfully. |
| TD_BADTA | An invalid internal process handle was passed in. |
| TD_BADPH | There is a `NULL` external process handle associated with this internal process handle. |
| TD_DBERR | A call to one of the imported interface routines failed. |
| TD_NOTHR | Either there is no thread with the given thread id ( `td_ta_map_id2thr` ) or no thread is currently executing on the given LWP ( `td_ta_map_lwp2thr` ). |
| TD_ERR | The call did not complete successfully. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

`libthread_db`(3THR) , `thr_create`(3THR) , `thr_self`(3THR) , `libthread_db`(3LIB) , `attributes`(5)

## NAME

td_ta_new, td_ta_delete, td_ta_get_ph – allocate and deallocate process handles for libthread_db

## SYNOPSIS

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]
#include <proc_service.h>
#include <thread_db.h>
td_err_e **td_ta_new**(const struct ps_prochandle **ph_p*, td_thragent_t ***ta_pp*);

td_err_e **td_ta_delete**(const td_thragent_t **ta_p*);

td_err_e **td_ta_get_ph**(const td_thragent_t **ta_p*, struct ps_prochandle ***ph_pp*);

## DESCRIPTION

`td_ta_new()` registers a target process with `libthread_db` and allocates an internal process handle of type `td_thragent_t` for this target process. Subsequent calls to `libthread_db` can use this handle to refer to this target process.

There are actually two process handles, an internal process handle assigned by `libthread_db` and an external process handle assigned by the `libthread_db` client. There is a one-to-one correspondence between the two handles. When the client calls a `libthread_db` routine, it uses the internal process handle. When `libthread_db` calls one of the client-provided routines listed in `proc_service`(3PROC) , it uses the external process handle.

*ph* is the external process handle that `libthread_db` should use to identify this target process to the controlling process when it calls routines in the imported interface.

If this call is successful, the value of the newly allocated `td_thragent_t` handle is returned in **ta_pp* . `td_ta_delete()` deregisters a target process with `libthread_db` , which deallocates its internal process handle and frees any other resources `libthread_db` has acquired with respect to the target process. *ta_p* specifies the target process to be deregistered.

`td_ta_get_ph()` returns in **ph_pp* the external process handle that corresponds to the internal process handle *ta_p* . This is useful for checking internal consistency.

## RETURN VALUES

| | |
|---|---|
| `TD_OK` | The call completed successfully. |
| `TD_BADPH` | A `NULL` external process handle was passed in to `td_ta_new` . |
| `TD_ERR` | `ta_pp` is `NULL,` or an internal error occurred. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |

| | |
|---|---|
| TD_MALLOC | Memory allocation failure. |
| TD_NOLIBTHREAD | The target process does not appear to be multithreaded. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

## SEE ALSO

libthread_db(3THR) , proc_service(3PROC) , libthread_db(3LIB) , attributes(5)

## NAME

td_ta_setconcurrency – set concurrency level for target process

## SYNOPSIS

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

```
#include &lt;proc_service.h&gt;
#include &lt;thread_db.h&gt;
td_err_e td_ta_setconcurrency(const td_thragent_t *ta_p, int level););
```

## DESCRIPTION

`td_ta_setconcurrency( )` sets the desired concurrency level for the process identified by *ta_p* to level, just as if a thread within the process had called `thr_setconcurrency( )`. See `thr_setconcurrency`(3THR).

## RETURN VALUES

`TD_OK` The call completed successfully.

`TD_BADTA` An invalid internal process handle was passed in.

`TD_BADPH` There is a `NULL` external process handle associated with this internal process handle. `TD_NOCAPAB` The client did not implement the `ps_kill( )` routine in the imported interface. See `ps_kill`(3PROC).

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_ERR` A `libthread_db` internal error occurred.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
| --- | --- |
| MT-Level | Safe |

## SEE ALSO

`libthread_db`(3THR), `ps_kill`(3PROC), `thr_setconcurrency`(3THR), `libthread_db`(3LIB), `attributes`(5)

**NAME**

td_ta_sync_iter, td_ta_thr_iter, td_ta_tsd_iter – iterator functions on process handles from libthread_db library of interfaces

**SYNOPSIS**

cc [ *flag* … ] *file* … -lthread_db [ *library* … ]

```
#include &lt;proc_service.h&gt;
#include &lt;thread_db.h&gt;
```

td_err_e **td_ta_sync_iter**(const td_thragent_t **ta_p*, td_sync_iter_f **cb*, void **cbdata_p*);

td_err_e **td_ta_tsd_iter**(const td_thragent_t **ta_p*, td_key_iter_f **cb*, void **cbdata_p*);

td_err_e **td_ta_sync_iter**(const td_thragent_t **ta_p*, td_sync_iter_f **cb*, void **cbdata_p*);

**DESCRIPTION**

`td_ta_sync_iter()` , `td_ta_thr_iter()` , and `td_ta_tsd_iter()` are iterator functions that when given a target process handle as an argument, return sets of handles for objects associated with the target process. The method is to call back a client-provided function once for each associated object, passing back a handle as well as the client-provided pointer *cb_data_p* . This enables a client to easily build a linked list of the associated objects.

`td_ta_sync_iter()` returns handles of synchronization objects (mutexes, preader-writer locks, semaphores, and condition variables) associated with a process. Some synchronization objects may not be known to `libthread_db()` and will not be returned. If the process has initialized the synchronization object (by calling `mutex_init()` , for example) or a thread in the process has blocked on this object after `libthread_db()` attached to the synchronization object, then a handle for the synchronization object will be returned by `libthread_db()` . See `td_sync_get_info`(3THR) to see operations that can be performed on synchronization object handles.

`td_ta_thr_iter()` returns handles for threads that are part of the target process. For `td_ta_thr_iter()` , the caller specifies several criteria to select a subset of threads for which the callback function should be called. Any of these selection criteria may be wild-carded. If all of them are wild-carded, then handles for all threads in the process will be returned.

The selection parameters and corresponding wild-card values are:

`state (TD_THR_ANY_STATE ):`
  Select only threads whose state matches `state` . See `td_thr_get_info`(3THR) for a list of thread states.

`ti_pri (TD_THR_LOWEST_PRIORITY ):`
  Select only threads for which the priority is at least `ti_pri` .

`ti_sigmask_p (TD_SIGNO_MASK ):`
  Select only threads whose signal mask exactly matches **ti_sigmask_p* .

`ti_user_flags (TD_THR_ANY_USER_FLAGS ):`

Select only threads whose user flags (specified at thread creation time) exactly match *ti_user_flags* .

`td_ta_tsd_iter( )` returns the thread-specific data keys in use by the current process. Thread-specific data for a particular thread and key may be obtained by calling `td_thr_tsd`(3THR) .

**RETURN VALUES**

`TD_OK` The call completed successfully.

`TD_BADTA` An invalid process handle was passed in.

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_ERR` The call did not complete successfully.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
| --- | --- |
| MT-Level | Safe |

**SEE ALSO**

`libthread_db`(3THR) , `td_sync_get_info`(3THR) , `td_thr_get_info`(3THR) , `td_thr_tsd`(3THR) , `libthread_db`(3LIB) , `attributes`(5)

## NAME

td_thr_dbsuspend, td_thr_dbresume – suspend and resume threads in libthread_db

## SYNOPSIS

cc [ *flag* … ] *file* … -lthread_db [ *library* … ]

```
#include &lt;proc_service.h&gt;
#include &lt;thread_db.h&gt;
td_err_e td_thr_dbsuspend(const td_thrhandle_t *th_p);

td_err_e td_thr_dbresume(const td_thrhandle_t *th_p);
```

## DESCRIPTION

These operations suspend and resume the thread identified by *th_p* . A thread that has been suspended with `td_thr_dbsuspend()` is said to be in the "dbsuspended" state. A thread whose "dbsuspended" flag is set will not execute. If an unbound thread enters the "dbsuspended" state and is currently assigned to a lightweight process ( LWP ), then the LWP becomes available for assignment to a different thread.

A thread's "dbsuspended" state is independent of the suspension state controlled by calls to `thr_suspend`(3THR) and `thr_continue`(3THR) from within the target process. Calling `thr_continue`(3THR) within the target process on a thread that has been suspended during a call to `td_thr_dbsuspend()` will not cause that thread to resume execution; only a call to `td_thr_dbresume()` will do that.

## RETURN VALUES

| | |
|---|---|
| `TD_OK` | The call completed successfully. |
| `TD_BADTH` | An invalid thread handle was passed in. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |
| `TD_NOCAPAB` | The "agent thread" in the target process has not completed initialization, so this operation cannot be performed. The operation can be performed after the target process has been allowed to make some forward progress. See also `libthread_db`(3THR) |
| `TD_ERR` | A `libthread_db` internal error occurred. |

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

libthread_db(3THR) , thr_continue(3THR) , thr_suspend(3THR) , libthread_db(3LIB) , attributes(5)

## NAME

td_thr_getgregs, td_thr_setgregs, td_thr_getfpregs, td_thr_setfpregs, td_thr_getxregsize, td_thr_getxregs, td_thr_setxregs – reading and writing thread registers in libthread_db

## SYNOPSIS

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>
td_err_e **td_thr_getgregs**(const td_thrhandle_t **th_p*, prgregset_t*gregset*);

td_err_e **td_thr_setgregs**(const td_thrhandle_t **th_p*, prgregset_t*gregset*);

td_err_e **td_thr_getfpregs**(const td_thrhandle_t **th_p*, prfpregset_t **fpregset*);

td_err_e **td_thr_setfpregs**(const td_thrhandle_t **th_p*, prfpregset_t **fpregset*);

td_err_e **td_thr_getxregsize**(const td_thrhandle_t **th_p*, int **xregsize*);

td_err_e **td_thr_getxregs**(const td_thrhandle_t **th_p*, prxregset_t **xregset*);

td_err_e **td_thr_setxregs**(const td_thrhandle_t **th_p*, prxregset_t **xregset*);

## DESCRIPTION

These routines read and write the register sets associated with thread *th_p* . `td_thr_getgregs()` and `td_thr_setgregs()` get and set, respectively, the general registers of thread *th_p* . `td_thr_getfpregs()` and `td_thr_setfpregs()` get and set, respectively, the thread's floating point register set. `td_thr_getxregsize()` , `td_thr_getxregs()` , and `td_thr_setxregs()` are SPARC-specific. `td_thr_getxregsize()` returns in * *xregsize* the size of the architecture-dependent extra state registers. `td_thr_getxregs()` and `td_thr_setxregs()` get and set, respectively, those extra state registers. On non-SPARC architectures, these routines return `TD_NOXREGS.`

If thread *th_p* is currently executing on a lightweight process ( LWP), these routines will read or write, respectively, the appropriate register set to the LWP using the imported interface. If the thread is not currently executing on a LWP, then the floating point and extra state registers may not be read or written. Some of the general registers may also not be readable or writable, depending on the architecture. In this case, `td_thr_getfpregs()` and `td_thr_setfpregs()` will return `TD_NOFPREGS,` and `td_thr_getxregs()` and `td_thr_setxregs()` will return `TD_NOXREGS.` Calls to `td_thr_getgregs()` and `td_thr_setgregs()` will succeed, but values returned for unreadable registers will be undefined, and values specified for unwritable registers will be ignored. In this instance, a value of `TD_PARTIALREGS` will be returned. See the architecture-specific notes that follow regarding the registers that may be read and written for a thread not currently executing on a LWP.

**SPARC** On a thread not currently assigned to a LWP, only %i0-%i7, %l0-%l7, %g7, %pc, and %sp (%o6) may be read or written. %pc and %sp refer to the program counter and stack pointer that the thread will have when it resumes execution.

**Intel IA** On a thread not currently assigned to a LWP, only %pc, %sp, %ebp, %edi, %edi, and %ebx may be read.

## RETURN VALUES

`TD_OK` The call completed successfully.

`TD_BADTH` An invalid thread handle was passed in.

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_PARTIALREGS` Because the thread is not currently assigned to a LWP, not all registers were read or written. See `DESCRIPTION` for a discussion about which registers are not saved when a thread is not assigned to an LWP.

`TD_NOFPREGS` Floating point registers could not be read or written, either because the thread is not currently assigned to an LWP, or because the architecture does not have such registers.

`TD_NOXREGS` Architecture-dependent extra state registers could not be read or written, either because the thread is not currently assigned to an LWP, or because the architecture does not have such registers, or because the architecture is not a SPARC architecture.

`TD_ERR` A `libthread_db` internal error occurred.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

## SEE ALSO

`libthread_db`(3THR) , `libthread_db`(3LIB) , `attributes`(5)

**NAME**

td_thr_get_info – get thread information in libthread_db library of interfaces

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

```
#include <proc_service.h>
#include <thread_db.h>
td_err_e td_thr_get_info(const td_thrhandle_t *th_p, td_thrinfo_t *ti_p);
```

**DESCRIPTION**

The `td_thr_get_info( )` routine fills in the `td_thrinfo_t` structure **ti_p* with values for the thread identified by *th_p*.

The `td_thrinfo_t` structure contains the following fields:

```
typedef struct td_thrinfo_t {
  td_thragen_tx    *ti_ta_p           /* internal process handle */
  unsigned         ti_user_flags;     /* value of flags parameter */
  thread_t         ti_tid;            /* thread identifier */
  char             *ti_tls;           /* pointer to thread-local storage*/
  paddr            ti_startfunc;      /* address of function at which thread
                                         execution began*/
  paddr            ti_stkbase;        /* base of thread's stack area*/
  int              ti_stksize;        /* size in bytes of thread's allocated
                                         stack region*/
  paddr            ti_ro_area;        /* address of uthread_t structure*/
  int              ti_ro_size         /* size of the uthread_t structure in
                                         bytes */
  td_thr_state_e   ti_state           /* state of the thread */
  uchar_t          ti_db_suspended    /* non-zero if thread suspended by
                                         td_thr_dbsuspend*/
  td_thr_type_e    ti_type            /* type of the thread*/
  int              ti_pc              /* value of thread's program counter*/
  int              ti_sp              /* value of thread's stack counter*/
  short            ti_flags           /* set of special flags used by
                                         libthread*/
  int              ti_pri             /* priority of thread returned by
                                         thr_getprio(3T)*/
  lwpid_t          ti_lid             /* id of light weight process (LWP)
                                         executing this thread*/
  sigset_t         ti_sigmask         /* thread's signal mask.  See
                                         thr_sigsetmask(3T)*/
  u_char           ti_traceme         /* non-zero if event tracing is on*/
  u_char_t         ti_preemptflag     /* non-zero if thread preempted when
                                         last active*/
  u_char_t         ti_pirecflag     /* non-zero if thread runs priority
                                       beside regular */
  sigset_t         ti_pending       /* set of signals pending for this
                                       thread*/
  td_thr_events_t  ti_events        /* bitmap of events enabled for this
                                       thread*/
} ;
```

`td_thragent_t` **ti_ta_p* is the internal process handle identifying the process of which the thread is a member.

unsigned ti_user_flags is the value of the flags parameter passed to thr_create(3THR) when the thread was created.

thread_t ti_tid is the thread identifier for the thread returned by libthread when created with thr_create(3THR).

char **ti_tls* is the thread's pointer to thread-local storage.

psaddr_t ti_startfunc is the address of the function at which thread execution began, as specified when the thread was created with thr_create(3THR).

psaddr_t ti_stkbase is the base of the thread's stack area.

int ti_stksize is the size in bytes of the thread's allocated stack region.

psaddr_t ti_ro_area is the address of the libthread-internal uthread_t structure for this thread. Since accessing the uthread_t structure directly violates the encapsulation provided by libthread_db, this field should generally not be used. However, it may be useful as a prototype for extensions.

td_thr_state_e ti_state is the state in which the thread is. The td_thr_state_e enumeration type may contain the following values:

| | |
|---|---|
| TD_THR_ANY_STATE | Never returned by td_thr_get_info. TD_THR_ANY_STATE is used as a wildcard to select threads in td_ta_thr_iter(). |
| TD_THR_UNKNOWN | libthread_db cannot determine the state of the thread. |
| TD_THR_STOPPED | The thread has been stopped by a call to thr_suspend(3THR). |
| TD_THR_RUN | The thread is runnable, but it is not currently assigned to a LWP. |
| TD_THR_ACTIVE | The thread is currently executing on a LWP. |
| TD_THR_ZOMBIE | The thread has exited, but it has not yet been deallocated by a call to thr_join(3THR). |
| TD_THR_SLEEP | The thread is not currently runnable. |
| TD_THR_STOPPED_ASLEEP | The thread is both blocked by TD_THR_SLEEP, and stopped by a call to td_thr_dbsuspend(3THR). |

`uchar_t` `ti_db_suspended` is non-zero if and only if this thread is currently suspended because the controlling process has called `td_thr_dbsuspend` on it.

`td_thr_type_e` `ti_type` is a type of thread. It will be either `TD_THR_USER` for a user thread (one created by the application), or `TD_THR_SYSTEM` for one created by `libthread`.

`int` `ti_pc` is the value of the thread's program counter, provided that the thread's `ti_state` value is `TD_THR_SLEEP`, `TD_THR_STOPPED`, or `TD_THR_STOPPED_ASLEEP`. Otherwise, the value of this field is undefined.

`int` `ti_sp` is the value of the thread's stack pointer, provided that the thread's `ti_state` value is `TD_THR_SLEEP`, `TD_THR_STOPPED`, or `TD_THR_STOPPED_ASLEEP`. Otherwise, the value of this field is undefined.

`short` `ti_flags` is a set of special flags used by `libthread`, currently of use only to those debugging `libthread`.

`int` `ti_pri` is the thread's priority, as it would be returned by `thr_getprio`(3THR).

`lwpid_t` `ti_lid` is the ID of the LWP executing this thread, or the ID of the LWP that last executed this thread, if this thread is not currently assigned to a LWP.

`sigset_t` `ti_sigmask` is this thread's signal mask. See `thr_sigsetmask`(3THR).

`u_char` `ti_traceme` is non-zero if and only if event tracing for this thread is on.

`uchar_t` `ti_preemptflag` is non-zero if and only if the thread was preempted the last time it was active.

`uchar_t` `ti_pirecflag` is non-zero if and only if due to priority inheritance the thread is currently running at a priority other than its regular priority.

`td_thr_events_t` `ti_events` is the bitmap of events enabled for this thread.

## RETURN VALUES

`TD_OK` The call completed successfully.

`TD_BADTH` An invalid thread handle was passed in.

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_ERR` The call did not complete successfully.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

libthread(3THR), libthread_db(3THR), td_ta_thr_iter(3THR), td_thr_dbsuspend(3THR), thr_create(3THR), thr_getprio(3THR), thr_join(3THR), thr_sigsetmask(3THR), thr_suspend(3THR), libthread(3LIB), libthread_db(3LIB), attributes(5)

**NAME**

td_thr_lockowner – iterate over the set of locks owned by a thread

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>
td_err_e **td_thr_lockowner**(const td_thrhandle_t **th_p*, td_sync_iter_f **cb*, void **cb_data_p*);

**DESCRIPTION**

`td_thr_lockowner()` calls the iterator function *cb* once for every mutex that is held by the thread whose handle is *th_p*. The synchronization handle and the pointer *cb_data_p* are passed to the function. See `td_ta_thr_iter`(3THR) for a similarly structured function.

Iteration terminates early if the callback function *cb* returns a non-zero value.

**RETURN VALUES**

| | |
|---|---|
| `TD_OK` | The call completed successfully. |
| `TD_BADTH` | An invalid thread handle was passed in. |
| `TD_BADPH` | There is a `NULL` external process handle associated with this internal process handle. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |
| `TD_ERR` | A `libthread_db` internal error occurred. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

`libthread_db`(3THR), `td_ta_thr_iter`(3THR), `libthread_db`(3LIB), `attributes`(5)

**NAME**

td_thr_setprio – set the priority of a thread

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

```
#include <proc_service.h>
#include <thread_db.h>
td_err_e td_thr_setprio(const td_thrhandle_t *th_p, const int new_prio;);
```

**DESCRIPTION**

`td_thr_setprio( )` sets thread *th_p*'s priority to *new_prio*, just as if a thread within the process had called `thr_setprio( ).` See `thr_setprio`(3THR).

**RETURN VALUES**

| | |
|---|---|
| `TD_OK` | The call completed successfully. |
| `TD_BADTH` | An invalid thread handle was passed in. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |
| `TD_ERR` | *new_prio* is an illegal value (out of range). |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

`libthread_db`(3THR), `thr_setprio`(3THR), `libthread_db`(3LIB), `attributes`(5)

**NAME**

td_thr_setsigpending, td_thr_sigsetmask – manage thread signals for libthread_db

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

#include <proc_service.h>
#include <thread_db.h>
td_err_e **td_thr_setsigpending**(const td_thrhandle_t * *th_p*, const uchar_ t *ti_sigpending_flag*, const sigset_t *ti_sigmask* ;);

td_err_e **td_thr_sigsetmask**(const td_thrhandle_t **th_p*, const sigset_t *ti_sigmask*);

**DESCRIPTION**

The `td_thr_setsigpending( )` and `td_thr_setsigmask( )` operations affect the signal state of the thread identified by *th_p* .

`td_thr_setsigpending( )` sets the set of pending signals for thread *th_p* to *ti_sigpending* . The value of the `libthread` -internal field that indicates whether a thread has any signal pending is set to *ti_sigpending_flag* . To be consistent, *ti_sigpending_flag* should be zero if and only if all of the bits in *ti_sigpending* are zero.

`td_thr_sigsetmask( )` sets the signal mask of the thread *th_p* as if the thread had set its own signal mask by way of `thr_sigsetmask`(3THR) . The new signal mask is the value of `ti_sigmask` .

There is no equivalent to the `SIG_BLOCK` or `SIG_UNBLOCK` operations of `thr_sigsetmask`(3THR) , which mask or unmask specific signals without affecting the mask state of other signals. To block or unblock specific signals, either stop the whole process, or the thread, if necessary, by `td_thr_dbsuspend( )` . Then determine the thread's existing signal mask by calling `td_thr_get_info( )` and reading the *ti_sigmask* field of the `td_thrinfo_t` structure returned. Modify it as desired, and set the new signal mask with `td_thr_sigsetmask( )` .

**RETURN VALUES**

| | |
|---|---|
| `TD_OK` | The call completed successfully. |
| `TD_BADTH` | An invalid thread handle was passed in. |
| `TD_DBERR` | A call to one of the imported interface routines failed. |
| `TD_ERR` | A `libthread_db` internal error occurred. |

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

libthread_db(3THR) , td_thr_dbsuspend(3THR) , td_thr_get_info(3THR) , libthread_db(3LIB) , attributes(5)

## NAME

td_thr_sleepinfo – return the synchronization handle for the object on which a thread is blocked

## SYNOPSIS

cc [ *flag* … ] *file* … -lthread_db [ *library* … ]

```
#include <proc_service.h>
#include <thread_db.h>
td_err_e td_thr_sleepinfo(const td_thrhandle_t *th_p, td_synchandle_t *sh_p);
```

## DESCRIPTION

`td_thr_sleepinfo()` returns in **sh_p* the handle of the synchronization object on which a sleeping thread is blocked.

## RETURN VALUES

`TD_OK` The call completed successfully.

`TD_BADTH` An invalid thread handle was passed in.

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_ERR` The thread *th_p* is not blocked on a synchronization object, or a `libthread_db` internal error occurred.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

## SEE ALSO

`libthread_db`(3THR), `libthread_db`(3LIB), `attributes`(5)

## NAME

td_thr_tsd – get a thread's thread-specific data for libthread_db library of interfaces

## SYNOPSIS

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]

```
#include <proc_service.h>
#include <thread_db.h>
td_err_e td_thr_tsd(const td_thrhandle_t, const thread_key_t key, void *data_pp);
```

## DESCRIPTION

`td_thr_tsd()` returns in **data_pp* the thread-specific data pointer for the thread identified by *th_p* and the thread-specific data key *key*. This is the same value that thread *th_p* would obtain if it called `thr_getspecific`(3THR).

To find all the thread-specific data keys in use in a given target process, call `td_ta_tsd_iter`(3THR).

## RETURN VALUES

`TD_OK` The call completed successfully.

`TD_BADTH` An invalid thread handle was passed in.

`TD_DBERR` A call to one of the imported interface routines failed.

`TD_ERR` A `libthread_db` internal error occurred.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

## SEE ALSO

`libthread_db`(3THR), `td_ta_tsd_iter`(3THR), `thr_getspecific`(3THR), `libthread_db`(3LIB), `attributes`(5)

**NAME**

td_thr_validate – test a thread handle for validity

**SYNOPSIS**

cc [ *flag* ... ] *file* ... -lthread_db [ *library* ... ]
#include <proc_service.h>
#include <thread_db.h>
td_err_e **td_thr_validate**(const td_thrhandle_t **th_p*);

**DESCRIPTION**

td_thr_validate( ) tests whether *th_p* is a valid thread handle. A valid thread handle may become invalid if its thread exits.

**RETURN VALUES**

TD_OK The call completed successfully. *th_p* is a valid thread handle.

TD_BADTH *th_p* was NULL.

TD_DBERR A call to one of the imported interface routines failed.

TD_NOTHR *th_p* is not a valid thread handle.

TD_ERR A libthread_db internal error occurred.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | Safe |

**SEE ALSO**

libthread_db(3THR), libthread_db(3LIB), attributes(5)

**NAME**

thr_create – create a tread

**SYNOPSIS**

cc – mt [ *flag*... ] *file*...[ *library*... ]

#include <thread.h>

int **thr_create**(void **stack_base*, size_t *stack_size*, void *(**start_func*) (void*), void **arg*, long *flags*, thread_t **new_thread_ID*);

**DESCRIPTION**

Thread creation adds a new thread of control to the current process. The procedure `main( )` is a single thread of control. Each thread executes simultaneously with all other threads within the calling process and with other threads from other active processes.

Although a newly created thread shares all of the calling process's global data with the other threads in the process, it has its own set of attributes and private execution stack. The new thread inherits the calling thread's signal mask, possibly, and scheduling priority. Pending signals for a new thread are not inherited and will be empty.

The call to create a thread takes the address of a user-defined function, specified by *start_func*, as one of its arguments. This function is the complete execution routine for the new thread.

The lifetime of a thread begins with the successful return from `thr_create( )`, which calls *start_func*( ) and ends with one of the following:

- the normal completion of *start_func*( ),
- the return from an explicit call to `thr_exit`(3THR), or
- the conclusion of the calling process (see `exit`(2)).

The new thread performs by calling the function defined by *start_func* with only one argument, *arg*. If more than one argument needs to be passed to *start_func*, the arguments can be packed into a structure, the address of which can be passed to *arg*.

If *start_func* returns, the thread terminates with the exit status set to the *start_func* return value (see `thr_exit`(3THR)).

When the thread from which `main( )` originated returns, the effect is the same as if an implicit call to `exit( )` were made using the return value of `main( )` as the exit status. This behavior differs from a *start_func* return. If `main( )` calls `thr_exit`(3THR), only the `main` thread exits, not the entire process.

If the thread creation fails, a new thread is not created and the contents of the location referenced by the pointer to the new thread are undefined.

The *flags* argument specifies which attributes are modifiable for the created thread. The value in *flags* is determined by the bitwise inclusive-OR of the following:

THR_BOUND This flag affects the contentionscope attribute of the thread. The new thread is created permanently bound to an LWP (that is, it is a *bound thread*). This thread will now contend among system-wide resources.

THR_DETACHED This flag affects the detachstate attribute of the thread. The new thread is created detached. The exit status of a detached thread is not accessible to other threads. Its thread ID and other resources may be re-used as soon as the thread terminates. thr_join(3THR) will not wait for a detached thread.

THR_NEW_LWP This flag affects the concurrency attribute of the thread. The desired concurrency level for unbound threads is increased by one. This is similar to incrementing concurrency by one by way of thr_setconcurrency(3THR) Typically, this adds a new LWP to the pool of LWPs running unbound threads.

THR_SUSPENDED This flag affects the suspended attribute of the thread. The new thread is created suspended and will not execute *start_func* until it is started by thr_continue( ).

THR_DAEMON This flag affects the daemon attribute of the thread. The thread is marked as a daemon. The process will exit when all non-daemon threads exit. thr_join(3THR) will not wait for a daemon thread. Daemon threads do not interfere with the exit conditions for a process. A process will terminate when all regular threads exit or the process calls exit( ). Daemon threads are most useful in libraries that want to use threads.

Default thread creation:

```
thread_t tid;
void *start_func(void *), *arg;
thr_create(NULL, NULL, start_func, arg, NULL, &tid);
```

User-defined thread creation (create a thread scheduled on a system-wide basis, that is, a bound thread):

```
thr_create(NULL, NULL, start_func, arg, THR_BOUND, &tid);
```

If both THR_BOUND and THR_NEW_LWP are specified, two LWPs are created, one for the bound thread and another for the pool of LWPs running unbound threads.

```
thr_create(NULL, NULL, start_func, arg, THR_BOUND | THR_NEW_LWP, &tid);
```

With `thr_create( )`, the new thread uses the stack beginning at the address specified by *stack_base* and continuing for *stack_size* bytes. The *stack_size* argument must be greater than the value returned by `thr_min_stack`(3THR). If *stack_base* is `NULL`, `thr_create( )` allocates a stack for the new thread with at least *stack_size* bytes. If *stack_size* is 0, a default size is used. If *stack_size* is not 0, it must be greater than the value returned by `thr_min_stack`(3THR) See `NOTES`.

When *new_thread_ID* is not `NULL`, it points to a location where the ID of the new thread is stored if `thr_create( )` is successful. The ID is only valid within the calling process.

**RETURN VALUES**

If successful, the `thr_create( )` function returns `0`. Otherwise, an error value is returned to indicate the error. If the application is not linked with the threads library, –`1` is returned.

**ERRORS**

The `thr_create( )` function will fail if:

`EAGAIN` The system-imposed limit on the total number of threads in a process has been exceeded or some system resource has been exceeded (for example, too many LWPs were created).

`EINVAL` The *stack_base* argument is not `NULL` and *stack_size* is less than the value returned by `thr_min_stack`(3THR), or the *stack_base* argument is NULL`NULL` and *stack_size* is not `0` and is less than the value returned by `thr_min_stack`(3THR).

The `thr_create( )` function may use `mmap( )` to allocate thread stacks from `MAP_PRIVATE`, `MAP_NORESERVE`, and `MAP_ANON` memory mappings if *stack_base* is `NULL`, and consequently may return upon failure the revelevant error values returned by `mmap( )`. See the `mmap`(2) manual page for these error values.

**EXAMPLES**

**EXAMPLE 1** This is an example of concurrency with multi-threading. Since POSIX threads and Solaris threads are fully compatible even within the same process, this example uses `pthread_create( )` if you execute `a.out 0`, or `thr_create( )` if you execute `a.out 1`.

Five threads are created that simultaneously perform a time-consuming function, `sleep(10)`. If the execution of this process is timed, the results will show that all five individual calls to sleep for ten-seconds completed in about ten seconds, even on a uniprocessor. If a single-threaded process calls `sleep(10)` five times, the execution time will be about 50-seconds.

The command-line to time this process is:

`/usr/bin/time a.out 0 (for POSIX threading)`

or

`/usr/bin/time a.out 1 (for Solaris threading)`

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic 3-lines for threads */
#include <pthread.h>
#include <thread.h>
#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);   /* thread routine */
int i;
thread_t tid[NUM_THREADS];      /* array of thread IDs */

int
main(int argc, char *argv[])
{
    if (argc == 1)  {
            printf("use 0 as arg1 to use pthread_create( )\n");
            printf("or use 1 as arg1 to use thr_create( )\n");
            return (1);
    }

    switch (*argv[1])  {
    case '0':  /* POSIX */
            for ( i = 0; i < NUM_THREADS; i++)
                    pthread_create(&tid[i], NULL, sleeping,
                        (void *)SLEEP_TIME);
            for ( i = 0; i < NUM_THREADS; i++)
                    pthread_join(tid[i], NULL);
            break;

    case '1':  /* Solaris */
            for ( i = 0; i < NUM_THREADS; i++)
                    thr_create(NULL, 0, sleeping, (void *)SLEEP_TIME, 0,
                        &tid[i]);
            while (thr_join(NULL, NULL, NULL) == 0)
                        ;
            break;
    }  /* switch */
    printf("main( ) reporting that all %d threads have terminated\n", i);
    return (0);
}  /* main */

void *
sleeping(void *arg)
{
    int sleep_time = (int)arg;
    printf("thread %d sleeping %d seconds ...\n", thr_self( ), sleep_time);
    sleep(sleep_time);
    printf("\nthread %d awakening\n", thr_self( ));
    return (NULL);
}
```

Had `main( )` not waited for the completion of the other threads (using `pthread_join`(3THR) or `thr_join`(3THR)), it would have continued to process concurrently until it reached the end of its routine and the entire process would have exited prematurely (see exit(2)).

**EXAMPLE 2** Creating a default thread with a new signal mask.

The following example demonstrates how to create a default thread with a new signal mask. The *new_mask* argument is assumed to have a value different from the creator's signal mask (*orig_mask*). The *new_mask* argument is set to block all signals except for SIGINT.. The creator's signal mask is changed so that the new thread inherits a different mask, and is restored to its original value after thr_create( ) returns.

This example assumes that SIGINT is also unmasked in the creator. If it is masked by the creator, then unmasking the signal opens the creator to this signal. The other alternative is to have the new thread set its own signal mask in its start routine.

```
thread_t tid;
sigset_t new_mask, orig_mask;
int error;

(void)sigfillset(&new_mask);
(void)sigdelset(&new_mask, SIGINT);
(void)thr_sigsetmask(SIG_SETMASK, &new_mask, &orig_mask):
error = thr_create(NULL, 0, do_func, NULL, 0, &tid);
(void)thr_sigsetmask(SIG_SETMASK, &orig_mask, NULL);
```

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

_lwp_create(2), exit(2), getrlimit(2), mmap(2), exit(3C), sleep(3C), thr_min_stack(3THR), thr_setconcurrency(3THR), thr_suspend(3THR), threads(3THR), attributes(5), standards(5)

## NOTES

MT application threads execute independently of each other, thus their relative behavior is unpredictable. Therefore, it is possible for the thread executing main( ) to finish before all other user application threads.

Using thr_join(3THR) in the following syntax,

while (thr_join(NULL, NULL, NULL) == 0);

will cause the invoking thread (which may be main( )) to wait for the termination of all other undetached and non-daemon threads; however, the second and third arguments to thr_join(3THR) need not necessarily be NULL.

A thread has not terminated until `thr_exit( )` has finished. The only way to determine this is by `thr_join( )`. When `thr_join( )` returns a departed thread, it means that this thread has terminated and its resources are reclaimable. For instance, if a user specified a stack to `thr_create( )`, this stack can only be reclaimed after `thr_join`( ) has reported this thread as a departed thread. It is not possible to determine when a *detached* thread has terminated. A detached thread disappears without leaving a trace.

Typically, thread stacks allocated by `thr_create( )` begin on page boundaries and any specified (a red-zone) size is rounded up to the next page boundary. A page with no access permission is appended to the top of the stack so that most stack overflows will result in a `SIGSEGV` signal being sent to the offending thread. Thread stacks allocated by the caller are used as is.

Using a default stack size for the new thread, instead of passing a user-specified stack size, results in much better `thr_create( )` performance. The default stack size for a user-thread is 1 megabyte in a 32-bit process and 2 megabyte in a 64-bit process.

A user-specified stack size must be greater than the value `THR_MIN_STACK.` A minimum stack size may not accommodate the stack frame for the user thread function *start_func*. If a stack size is specified, it must accommodate *start_func* requirements and the functions that it may call in turn, in addition to the minimum requirement.

It is usually very difficult to determine the runtime stack requirements for a thread. `THR_MIN_STACK` specifies how much stack storage is required to execute a NULL *start_func*. The total runtime requirements for stack storage are dependent on the storage required to do runtime linking, the amount of storage required by library runtimes (like `printf( )`) that your thread calls. Since these storage parameters are not known before the program runs, it is best to use default stacks. If you know your runtime requirements or decide to use stacks that are larger than the default, then it makes sense to specify your own stacks.

**NAME**

threads, pthreads, libpthread, libthread – concepts related to POSIX pthreads and Solaris threads and the libpthread and libthread libraries

**SYNOPSIS**

**POSIX**

cc -mt [ *flag...* ] *file...*- lpthread [ -lposix4 *library...* ]

#include <pthread.h>

**Solaris**

cc - mt [ *flag...* ] *file...*[ *library...* ]

#include <sched.h>

#include <thread.h>

**DESCRIPTION**

POSIX and Solaris threads each have their own implementation of the threads library. The `libpthread` library is associated with POSIX; the `libthread` library is associated with Solaris. Both implementations are interoperable, their functionality similar, and can be used within the same application. Only POSIX threads are guaranteed to be fully portable to other POSIX-compliant environments. POSIX and Solaris threads require different source, include files and linking libraries. See SYNOPSIS .

**Similarities**

Most of the functions in the `libpthread` and `libthread` , libraries have a counterpart in the other corresponding library. POSIX function names, with the exception of the semaphore names, have a "`pthread` " prefix. Function names for similar POSIX and Solaris have similar endings. Typically, similar POSIX and Solaris functions have the same number and use of arguments.

**Differences**

POSIX pthreads and Solaris threads differ in the following ways:

- POSIX threads are more portable.
- POSIX threads establish characteristics for each thread according to configurable attribute objects.
- POSIX pthreads implement thread cancellation.
- POSIX pthreads enforce scheduling algorithms.
- POSIX pthreads allow for clean-up handlers for `fork`(2) calls.
- Solaris threads can be suspended and continued.
- Solaris threads implement an optimized mutex and interprocess robust mutex locks.
- Solaris threads implement daemon threads, for whose demise the process does not wait.

**Function Comparison**

The following table compares the POSIX pthreads and Solaris threads functions. When a comparable interface is not available either in POSIX pthreads or Solaris threads, a hyphen (- ) appears in the column.

**Functions Related to Creation**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_create() | thr_create() |
| pthread_attr_init() | - |
| pthread_attr_setdetachstate() | - |
| pthread_attr_getdetachstate() | - |
| pthread_attr_setinheritsched() | - |
| pthread_attr_getinheritsched() | - |
| pthread_attr_setschedparam() | - |
| pthread_attr_getschedparam() | - |
| pthread_attr_setschedpolicy() | - |
| pthread_attr_getschedpolicy() | - |
| pthread_attr_setscope() | - |
| pthread_attr_getscope() | - |
| pthread_attr_setstackaddr() | - |
| pthread_attr_getstackaddr() | - |
| pthread_attr_setstacksize() | - |
| pthread_attr_getstacksize() | - |
| pthread_attr_getguardsize() | - |
| pthread_attr_setguardsize() | - |
| pthread_attr_destroy() | - |
| - | thr_min_stack() |

**Functions Related to Exit**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_exit() | thr_exit() |
| pthread_join() | thr_join() |
| pthread_detach() | - |

**Functions Related to Thread Specific Data**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_key_create( ) | thr_keycreate( ) |
| pthread_setspecific( ) | thr_setspecific( ) |
| pthread_getspecific( ) | thr_getspecific( ) |
| pthread_key_delete( ) | - |

**Functions Related to Signals**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_sigmask( ) | thr_sigsetmask( ) |
| pthread_kill( ) | thr_kill( ) |

**Functions Related to IDs**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_self( ) | thr_self( ) |
| pthread_equal( ) | - |
| - | thr_main( ) |

**Functions Related to Scheduling**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| - | thr_yield( ) |
| - | thr_suspend( ) |
| - | thr_continue( ) |
| pthread_setconcurrency( ) | thr_setconcurrency( ) |
| pthread_getconcurrency( ) | thr_getconcurrency( ) |
| pthread_setschedparam( ) | thr_setprio( ) |
| pthread_getschedparam( ) | thr_getprio( ) |

**Functions Related to Cancellation**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_cancel( ) | - |
| pthread_setcancelstate( ) | - |
| pthread_setcanceltype( ) | - |
| pthread_testcancel( ) | - |
| pthread_cleanup_pop( ) | - |
| pthread_cleanup_push( ) | - |

**Functions Related to Mutexes**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_mutex_init( ) | mutex_init( ) |
| pthread_mutexattr_init( ) | - |
| pthread_mutexattr_setpshared( ) | - |
| pthread_mutexattr_getpshared( ) | - |
| pthread_mutexattr_setprotocol( ) | - |
| pthread_mutexattr_getprotocol( ) | - |
| pthread_mutexattr_setprioceiling( ) | - |
| pthread_mutexattr_getprioceiling( ) | - |
| pthread_mutexattr_settype( ) | - |
| pthread_mutexattr_gettype( ) | - |
| pthread_mutexattr_destroy( ) | - |
| pthread_mutex_setprioceiling( ) | - |
| pthread_mutex_getprioceiling( ) | - |
| pthread_mutex_lock( ) | mutex_lock( ) |
| pthread_mutex_trylock( ) | mutex_trylock( ) |
| pthread_mutex_unlock( ) | mutex_unlock( ) |
| pthread_mutex_destroy( ) | mutex_destroy( ) |

**Functions Related to Condition Variables**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_cond_init( ) | cond_init( ) |
| pthread_condattr_init( ) | - |
| pthread_condattr_setpshared( ) | - |
| pthread_condattr_getpshared( ) | - |
| pthread_condattr_destroy( ) | - |
| pthread_cond_wait( ) | cond_wait( ) |
| pthread_cond_timedwait( ) | cond_timedwait( ) |
| pthread_cond_signal( ) | cond_signal( ) |
| pthread_cond_broadcast( ) | cond_broadcast( ) |
| pthread_cond_destroy( ) | cond_destroy( ) |

**Functions Related to Reader/Writer Locking**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_rwlock_init( ) | rwlock_init( ) |
| pthread_rwlock_rdlock( ) | rw_rdlock( ) |
| pthread_rwlock_tryrdlock( ) | rw_tryrdlock( ) |
| pthread_rwlock_wrlock() | rw_wrlock( ) |
| pthread_rwlock_trywrlock( ) | rw_trywrlock( ) |
| pthread_rwlock_unlock( ) | rw_unlock( ) |
| pthread_rwlock_destroy( ) | rwlock_destroy( ) |
| pthread_rwlockattr_init( ) | - |
| pthread_rwlockattr_destroy( ) | - |
| pthread_rwlockattr_getpshared( ) | - |
| pthread_rwlockattr_setpshared( ) | - |

**Functions Related to Semaphores**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| sem_init( ) | sema_init( ) |
| sem_open( ) | - |
| sem_close( ) | - |
| sem_wait( ) | sema_wait( ) |
| sem_trywait( ) | sema_trywait( ) |
| sem_post( ) | sema_post( ) |
| sem_getvalue( ) | - |
| sem_unlink( ) | - |
| sem_destroy( ) | sema_destroy( ) |

**Functions Related to fork() Clean Up**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_atfork( ) | - |

**Functions Related to Limits**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| pthread_once( ) | - |

**Functions Related to Debugging**

| POSIX (libpthread) | Solaris (libthread) |
|---|---|
| - | thr_stksegment( ) |

## LOCKING

**Synchronization**

POSIX (libpthread) Solaris (libthread) Multi-threaded behavior is asynchronous, and therefore, optimized for concurrent and parallel processing. As threads, always from within the same process and sometimes from multiple processes, share global data with each other, they are not guaranteed exclusive access to the shared data at any point in time. Securing mutually exclusive access to shared data requires synchronization among the threads. Both POSIX and Solaris implement four synchronization mechanisms: mutexes, condition variables, reader / writer locking (*optimized frequent-read occasional-write mutex* ), and semaphores.

Synchronizing multiple threads diminishes their concurrency. The coarser the grain of synchronization, that is, the larger the block of code that is locked, the lesser the concurrency.

**MT fork()**

If a POSIX threads program calls `fork`(2) , it implicitly calls `fork1`(2) , which replicates only the calling thread. Should there be any outstanding mutexes throughout the process, the application should call `pthread_atfork`(3THR) , to wait for and acquire those mutexes, prior to calling `fork()` .

## SCHEDULING

**POSIX**

Scheduling allocation size per thread is greater than one. POSIX supports the following three scheduling policies:

`SCHED_OTHER` Timesharing (TS) scheduling policy. It is based on the timesharing scheduling class.

`SCHED_FIFO` First-In-First-Out (FIFO) scheduling policy. Threads scheduled to this policy, if not pre-empted by a higher priority, will proceed until completion. Threads whose contention scope is system (`PTHREAD_SCOPE_SYSTEM` ) are in real-time (RT) scheduling class. The calling process must have a effective user ID of `0` . `SCHED_FIFO` for threads whose contention scope's process (`PTHREAD_SCOPE_PROCESS` ) is based on the TS scheduling class.

`SCHED_RR` Round-Robin scheduling policy. Threads scheduled to this policy, if not pre-empted by a higher priority, will execute for a time period determined by the system. Threads whose contention scope is system (`PTHREAD_SCOPE_SYSTEM` ) are in real-time (RT) scheduling class and the calling process must have a effective user ID of `0` . `SCHED_RR` for threads whose contention scope is process (`PTHREAD_SCOPE_PROCESS` ) is based on the TS scheduling class.

**Solaris**

Only scheduling policy supported is `SCHED_OTHER,` which is timesharing, based on the TS scheduling class.

## ALTERNATE IMPLEMENTATION

The standard threads implementation is a two-level model in which user-level threads are multiplexed over possibly fewer lightweight processes, or LWP s. An LWP is the fundamental unit of execution that is dispatched to a processor by the operating system.

The system provides an alternate threads implementation, a one-level model, in which user-level threads are associated one-to-one with LWP s. This implementation is simpler than the standard implementation and may be beneficial to some multithreaded applications. It provides exactly the same interfaces, both for POSIX threads and Solaris threads, as the standard implementation.

To link with the alternate implementation, use the following runpath (–R ) options when linking the program:

**POSIX**

```
cc -mt ... -lpthread ... -R /usr/lib/lwp     (32-bit)
cc -mt ... -lpthread ... -R /usr/lib/lwp/64  (64-bit)
```

**Solaris**

```
cc -mt ... -R /usr/lib/lwp     (32-bit)
cc -mt ... -R /usr/lib/lwp/64  (64-bit)
```

For multithreaded programs that have been previously linked with the standard threads library, the environment variables LD_LIBRARY_PATH and LD_LIBRARY_PATH_64 can be set as follows to bind the program at runtime to the alternate threads library:

```
LD_LIBRARY_PATH=/usr/lib/lwp
LD_LIBRARY_PATH_64=/usr/lib/lwp/64
```

Note that if an LD_LIBRARY_PATH environment variable is in effect for a secure process, then only the trusted directories specified by this variable will be used to augment the runtime linker's search rules.

The runtime linker may also be instructed to use this libthread by establishing an alternative object cache; see crle(1) with the –a option.

When using the alternate one-level threads implementation, be aware that it may create more LWP s than the standard implementation using unbound threads. LWP s consume operating system memory in contrast to threads, which consume only user-level memory. Thus a multithreaded application linked against this library that creates thousands of threads would create an equal number of LWP s and might run the system out of resources required to support the application.

## ERRORS

In a multi-threaded application, linked with libpthread or libthread , EINTR may be returned whenever another thread calls fork(2) , which calls fork1(2) instead.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe, Fork 1-Safe |

## FILES

**POSIX**

/usr/include/pthread.h /lib/libpthread.* /lib/libposix4.*

**Solaris**

/usr/include/thread.h /usr/include/sched.h /lib/libthread.*

## SEE ALSO

crle(1) , fork(2) , pthread_atfork(3THR) , pthread_create(3THR) , attributes(5) , standards(5)

Linker and Libraries Guide

**NAME**

thr_exit – terminate the calling thread

**SYNOPSIS**

cc –mt [ *flag...* ] *file...*[ *library...* ]

#include <thread.h>
void **thr_exit**(void **status*);

**DESCRIPTION**

`thr_exit( )` terminates the calling thread, in a similar way that `exit`(3C) terminates the calling process. If the calling thread is not detached, then the thread's ID and the exit status specified by *status* are retained. The value *status* is then made available to any successful join with the terminating thread (see `thr_join`(3THR)); otherwise, *status* is disregarded allowing the thread's ID to be reclaimed immediately.

Any cancellation cleanup handlers that have been pushed and not yet popped are popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions will be called in an unspecified order. Thread termination does not release any application visible process resources,including, but not limited to, mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to, calling any `atexit( )` routines that may exist.

If any thread, including the `main( )` thread, calls `thr_exit( )`, only that thread will exit.

If `main( )` returns or exits (either implicitly or explicitly), or any thread explicitly calls `exit( )`, the entire process will exit.

The behavior of `thr_exit( )` is undefined if called from a cancellation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to `thr_exit( )`.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the `thr_exit( )` *status* parameter value.

The process exits with an exit status of `0` after the last thread has been terminated. The behavior is as if the implementation called `exit( )` with a `0` argument at thread termination time.

If any thread (except the `main( )` thread) implicitly or explicitly returns, the result is the same as if the thread called `thr_exit( )` and it will return the value of *status* as the exit code.

The process will terminate with an exit status of `0` after the last thread has terminated (including the `main( )` thread). This action is the same as if the application had called `exit( )` with a `0` argument at thread termination time.

**RETURN VALUES** The `thr_exit()` function cannot return to its caller.

**ERRORS** No errors are defined.

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO** `exit`(3C), `thr_create`(3THR), `thr_join`(3THR), `thr_keycreate`(3THR), `attributes`(5), `standards`(5)

**NOTES** Although only POSIX implements cancellation, cancellation can be used with Solaris threads, due to their interoperability.

*status* should not reference any variables local to the calling thread.

**NAME**

thr_getconcurrency, thr_setconcurrency – get or set thread concurrency level

**SYNOPSIS**

cc -mt [ *flag...* ] *file*...[ *library...* ]

```
#include <thread.h>
int thr_setconcurrency(int new_level);

int thr_getconcurrency(void););
```

**DESCRIPTION**

Unbound threads in a process may or may not be required to be simultaneously active. See `thr_create`(3THR) . By default, the threads system ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency. `thr_setconcurrency( )` permits the application to give the threads system a hint, specified by *new_level* , for the desired level of concurrency. The actual number of simultaneously active threads may be larger or smaller than this number. The value for the desired concurrency level may also be affected by creating threads with the `THR_NEW_LWP` flag set. See `thr_create`(3THR) .

If *new_level* is `0` , the threads system will only ensure that a sufficient number of threads are active so that the process can continue to make progress.

`thr_getconcurrency( )` returns the current value for the desired concurrency level. The actual number of simultaneously active threads may be larger or smaller than this number.

**RETURN VALUES**

The `thr_getconcurrency( )` function always returns the current value for the desired concurrency level.

If successful, the `thr_setconcurrency( )` function returns `0` . Otherwise, a non-zero value is returned to indicate the error.

**ERRORS**

The `thr_setconcurrency( )` function will fail if:

`EAGAIN` The specified concurrency level would cause a system resource to be exceeded.

`EINVAL` *new_level* is negative.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`thr_create`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

thr_getprio, thr_setprio – access dynamic thread scheduling

**SYNOPSIS**

cc -mt [ *flag...* ] *file...*[ *library...* ]

```
#include <thread.h>
int thr_setprio(thread_t target_thread, int priority);

int thr_getprio(thread_t target_thread, int *priority);
```

**DESCRIPTION**

Thread scheduling is controlled by three attributes: its scope of contention, being either inter-process or intra-process (bound vs. unbound), (see `priocntl`(2) ); a relative scheduling priority; and a scheduling policy.

**Contentionscope**

Bound threads, which are inter-process, compete system-wide for scheduling resources and must be set at creation, for example:

```
thr_create(NULL,NULL, thread_routine, arg, THR_BOUND, NULL);
```

A bound thread is bound to an LWP and its scheduling is dependent upon the scheduling of the LWP to which it is bound. LWPs compete with other LWPs in other processes, however, their scheduling may be dynamically controlled by `priocntl`(2) .

By default, the scope for newly-created threads are unbound, or intra-process, and their setting is `NULL.` An unbound thread is scheduled by `libthread` on an underlying LWP, which competes with other LWPs in the same process.

The following dynamic scheduling functions should be used only with unbound threads: `thr_setprio( )` , and `thr_getprio( ).`

**Priority**

Priority scheduling is determined as follows:

- Higher priority threads are scheduled before lower priority threads.
- Solaris threads assumes that the priority is inherited across a thread create.
- A Solaris thread can be created suspended and its priority can be modified.

`thr_setprio( )` can dynamically modify an unbound thread's priority, and `thr_getprio( )` can read an unbound thread's priority.

**Policy**

The scheduling *policy* setting is:

`SCHED_OTHER` *(system default, often time-sharing)* Competing threads in this class are multiplexed according to their relative *priority* .

**Scheduling**

Solaris scheduling may only dynamically affect *priority* . There is no functionality to alter the *policy* of any thread; by default, a Solaris thread's schedule is equivalent to `SCHED_OTHER,` which is the only available Solaris policy.

`thr_setprio( )` changes the priority of the thread, specified by *target_thread* , within the current process to the priority specified by *priority* . Currently, by default, threads are scheduled based on fixed priorities that range from zero, the least significant, to 127. The *target_thread* will preempt lower priority threads, and will yield to higher priority threads in their contention for LWPs, not CPUs.

The function `thr_getprio( )` stores the current priority for the thread specified by *target_thread* in the location pointed to by *priority* . Note that thread priorities regulate access to LWPs, not CPUs, and hence are different from real-time priorities, which regulate and enforce access to CPU resources. A thread's priority set via these functions is more like a hint in terms of guaranteed access to execution resources. Programs that need access to "real" priorities should use bound threads in the real-time class (see `priocntl`(2) ).

**RETURN VALUES**

If successful, the `thr_getprio( )` and `thr_setprio( )` return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

For each of the following conditions, these functions return an error number if the condition is detected.

`ESRCH` The value specified by *target_thread* does not refer to an existing thread.

The `thr_getprio( )` and `thr_setprio( )` functions may fail if:

`EINVAL` The value of *priority* makes no sense for the scheduling class associated with the *target_thread.*

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`priocntl`(2) , `sched_setparam`(3RT) , `thr_create`(3THR) , `thr_suspend`(3THR) , `thr_yield`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

thr_join – wait for thread termination

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...[ *library*... ]

```
#include <thread.h>
int thr_join(thread_t thread, thread_t *departed, void **status);
```

**DESCRIPTION**

The `thr_join( )` functions suspend processing of the calling thread until the target *thread* completes. *thread* must be a member of the current process and it cannot be a detached or daemon thread. See `thr_create`(3THR).

Several threads cannot wait for the same thread to complete; one thread will complete successfully and the others will terminate with an error of `ESRCH`. `thr_join( )` will not block processing of the calling thread if the target *thread* has already terminated.

`thr_join( )` returns successfully when the target *thread* terminates.

If a `thr_join( )` call returns successfully with a non-null *status* argument, the value passed to `thr_exit`(3THR) by the terminating thread will be placed in the location referenced by *status*.

If the target *thread* ID is `0`, `thr_join( )` waits for any undetached thread in the process to terminate.

If *departed* is not `NULL`, it points to a location that is set to the ID of the terminated thread if `thr_join( )` returns successfully.

**RETURN VALUES**

If successful, `thr_join( )` returns `0`. Otherwise, an error number is returned to indicate the error.

**ERRORS**

`ESRCH` No undetached thread could be found corresponding to that specified by the given thread ID.

`EDEADLK` A recursive deadlock was detected, the value of *thread* specifies the calling thread. See NOTES.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`wait`(2), `thr_create`(3THR), `thr_exit`(3THR), `attributes`(5), `standards`(5)

**NOTES**

Using `thr_join`(3THR) in the following syntax,

```
while (thr_join(NULL, NULL, NULL) == 0);
```

will wait for the termination of all other undetached and non-daemon threads; after which, EDEADLK will be returned.

## NAME

thr_keycreate, thr_setspecific, thr_getspecific – thread-specific-data functions

## SYNOPSIS

cc -mt [ *flag...* ] *file*...[ *library...* ]

```
#include <thread.h>
int thr_keycreate(thread_key_t *keyp, void (*destructor, void *value);

int thr_setspecific(thread_key_t key, void *value);

int thr_getspecific(thread_key_t key, void **valuep);
```

## DESCRIPTION

### Create Key

In general, thread key creation allocates a key that locates data specific to each thread in the process. The key is global to all threads in the process, which allows each thread to bind a value to the key once the key has been created. The key independently maintains specific values for each binding thread. The `thr_keycreate( )` function allocates a global *key* namespace, pointed to by *keyp* , that is visible to all threads in the process. Each thread is initially bound to a private element of this *key* , which allows access to its thread-specific data.

Upon key creation, a new key is assigned the value `NULL` for all active threads. Additionally, upon thread creation, all previously created keys in the new thread are assigned the value `NULL.`

Optionally, a destructor function, *destructor* , may be associated with each *key* . Upon thread exit, if a *key* has a non-NULL *destructor* function and the thread has a non-NULL *value* associated with that *key* , the *destructor* function is called with the current associated *value* . If more than one *destructor* exists for a thread when it exits, the order of destructor calls is unspecified.

### Set Value

Once a key has been created, each thread may bind a new *value* to the key using `thr_setspecific( )` . The values are unique to the binding thread and are individually maintained. These values continue for the life of the calling thread.

Proper synchronization of *key* storage and access must be ensured by the caller. The *value* argument to `thr_setspecific( )` is generally a pointer to a block of dynamically allocated memory reserved by the calling thread for its own use. See `EXAMPLES` .

At thread exit, the *destructor* function, which is associated at time of creation, is called and it uses the specific key value as its sole argument.

### Get Value

`thr_getspecific( )` stores the current value bound to *key* for the calling thread into the location pointed to by *valuep* .

## RETURN VALUES

If successful, `thr_keycreate( )` , `thr_setspecific( )` and `thr_getspecific( )` return `0` . Otherwise, an error number is returned to indicate the error.

**ERRORS**

If the following conditions occur, `thr_keycreate( )` returns the corresponding error number:

`EAGAIN` The system lacked the necessary resources to create another thread-specific data key.

`ENOMEM` Insufficient memory exists to create the key.

If the following conditions occur, `thr_keycreate( )` and `thr_setspecific( )` return the corresponding error number:

`ENOMEM` Insufficient memory exists to associate the value with the key.

The `thr_setspecific( )` function returns the corresponding error number:

`EINVAL` The *key* value is invalid.

**EXAMPLES**

**EXAMPLE 1** In this example, the thread-specific data in this function can be called from more than one thread without special initialization.

For each argument you pass to the executable of this example, a thread is created and privately bound to the string-value of that argument.

```
/* cc thisfile.c */

#define _REENTRANT
#include <thread.h>
void *thread_specific_data(), free();
#define MAX_ARGC 20
thread_t tid[MAX_ARGC];
int num_threads;

main( int argc, char *argv[] ) {
  int i;
  num_threads = argc - 1;
  for( i = 0; i < num_threads; i++)
     thr_create(NULL, 0, thread_specific_data, argv[i+1]);
  for( i = 0; i < num_threads; i++)
     thr_join(tid[i], NULL, NULL);
}  /* end main */

void *thread_specific_data(char private_data[])
{
  static mutex_tkeylock; /* static ensures only one copy of keylock */
  static thread_key_tkey;
  static intonce_per_keyname = 0;
  void *tsd = NULL;

  if (!once_per_keyname) {
       mutex_lock(&keylock);
       if (!once_per_keyname)  {
             thr_keycreate(&key, free);
             once_per_keyname++;
```

```
    }
         mutex_unlock(&keylock);
    }
    tsd = thr_getspecific(key);
    if (tsd == NULL) {
         tsd = (void *)malloc(strlen(private_data) + 1);
         strcpy(tsd, private_data);
         thr_setspecific(key, tsd);
         printf("tsd for %d = %s\
",thr_self(),(char *)thr_getspecific(key));
         sleep(2);
         printf("tsd for %d remains %s\
",thr_self(),(char *)thr_getspecific(key));
    }
}  /* end thread_specific_data */

void
free(void *v)   {
    /* application-specific clean-up function */
}
```

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

thr_exit(3THR) , attributes(5) , standards(5)

**WARNINGS**

The thr_getspecific( ) and thr_getspecific( ) functions may be called either explicitly, or implicitly from a thread-specific data destructor function. Calling thr_setspecific( ) from a destructor may result in lost storage or infinite loops.

**NAME**

thr_kill – send a signal to a thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...[ *library*... ]

```
#include <signal.h>
#include <thread.h>
int thr_kill(thread_t thread, int sig);
```

**DESCRIPTION**

`thr_kill( )` sends the *sig* signal to the thread designated by *thread*. *thread* must be a member of the same process as the calling thread. *sig* must be one of the signals listed in `signal`(3HEAD); with the exception of `SIGLWP, SIGCANCEL,` and `SIGWAITING` being reserved and off limits to `thr_kill( )`. If *sig* is `0`, a validity check is done for the existence of the target thread; no signal is sent.

**RETURN VALUES**

Upon successful completion, `thr_kill( )` returns `0`. Otherwise, an error number is returned. In the event of failure, no signal is sent.

**ERRORS**

`ESRCH` No thread was found that corresponded to the thread designated by *thread* ID.

`EINVAL` The *sig* argument value is not zero and is an invalid or an unsupported signal number.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`kill`(2), `sigaction`(2), `raise`(3C), `thr_self`(3THR), `attributes`(5), `signal`(3HEAD), `standards`(5)

**NAME**

thr_main – identify the main thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...[ *library*... ]

```
#include <thread.h>
int thr_main(void););
```

**DESCRIPTION**

The `thr_main()` function returns one of the following:

`1` if the calling thread is the main thread

`0` if the calling thread is not the main thread

`-1` if `libthread` is not linked in or thread initialization has not completed

**FILES**

`/lib/libthread`

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`thr_self`(3THR), `attributes`(5)

**NAME**

thr_min_stack – return the minimum-allowable size for a thread's stack

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...[ *library*... ]

#include <thread.h>
size_t **thr_min_stack**(*void*);

**DESCRIPTION**

When a thread is created with a user-supplied stack, the user must reserve enough space to run this thread. In a dynamically linked execution environment, it is very hard to know what the minimum stack requirments are for a thread. The function `thr_min_stack( )` returns the amount of space needed to execute a null thread. This is a thread that was created to execute a null procedure. A thread that does something useful should have a stack size that is `thr_min_stack( )` + <*some increment*>.

Most users should not be creating threads with user-supplied stacks. This functionality was provided to support applications that wanted complete control over their execution environment.

Typically, users should let the threads library manage stack allocation. The threads library provides default stacks which should meet the requirements of any created thread.

`thr_min_stack( )` will return the unsigned int `THR_MIN_STACK`, which is the minimum-allowable size for a thread's stack.

In this implementation the default size for a user-thread's stack is one mega-byte. If the second argument to `thr_create`(3THR) is NULL, then the default stack size for the newly-created thread will be used. Otherwise, you may specify a stack-size that is at least `THR_MIN_STACK`, yet less than the size of your machine's virtual memory.

It is recommended that the default stack size be used.

To determine the smallest-allowable size for a thread's stack, execute the following:

```
/* cc thisfile.c -lthread */
#define _REENTRANT
#include <thread.h>
#include <stdio.h>
main( )  {
        printf("thr_min_stack( ) returns %u\n",thr_min_stack( ));
}
```

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO** attributes(5), standards(5)

**NAME**

thr_self – get calling thread's ID

**SYNOPSIS**

cc –mt [ *flag...* ] *file*...[ *library...* ]
#include <thread.h>
thread_t **thr_self**(void));

typedef(unsigned int thread_t);

**DESCRIPTION**

`thr_self()` returns the thread ID of the calling thread.

**ERRORS**

No errors are defined.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

`thr_create`(3THR), `attributes`(5), `standards`(5)

**NAME**

thr_sigsetmask – change or examine calling thread's signal mask

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...[ *library*... ]
#include <thread.h>
#include <signal.h>
int **thr_sigsetmask**(int *how*, const sigset_t **set*, sigset_t **oset*);

**DESCRIPTION**

The `thr_sigsetmask( )` function changes or examines a calling thread's signal mask. Each thread has its own signal mask. A new thread inherits the calling thread's signal mask and priority; however, pending signals are not inherited. Signals pending for a new thread will be empty.

If the value of the argument `set` is not `NULL`, `set` points to a set of signals that can modify the currently blocked set. If the value of `set` is `NULL`, the value of *how* is insignificant and the thread's signal mask is unmodified; thus, `thr_sigsetmask( )` can be used to inquire about the currently blocked signals.

The value of the argument *how* specifies the method in which the set is changed and takes one of the following values:

`SIG_BLOCK` `set` corresponds to a set of signals to block. They are added to the current signal mask.

`SIG_UNBLOCK` `set` corresponds to a set of signals to unblock. These signals are deleted from the current signal mask.

`SIG_SETMASK` `set` corresponds to the new signal mask. The current signal mask is replaced by `set`.

If the value of *oset* is not `NULL`, it points to the location where the previous signal mask is stored.

**RETURN VALUES**

Upon successful completion, the `thr_sigsetmask( )` function returns `0`. Otherwise, it returns a non-zero value.

**ERRORS**

The `thr_sigsetmask( )` function will fail if:

`EINVAL` The value of *how* is not defined and *oset* is `NULL`.

**EXAMPLES**

**EXAMPLE 1** The following example shows how to create a default thread that can serve as a signal catcher / handler with its own signal mask. `new` will have a different value from the creator's signal mask.

As POSIX threads and Solaris threads are fully compatible even within the same process, this example uses `pthread_create`(3THR) if you execute `a.out 0`, or `thr_create`(3THR) if you execute `a.out 1`.

In this example:

- `sigemptyset`(3C) initializes a null signal set, `new`. `sigaddset`(3C) packs the signal, `SIGINT`, into that new set.

- Either `pthread_sigmask( )` or `thr_sigsetmask( )` is used to mask the signal, `SIGINT` (CTRL-C), from the calling thread, which is `main( )`. The signal is masked to guarantee that only the new thread will receive this signal.
- `pthread_create( )` or `thr_create( )` creates the signal-handling thread.
- Using `pthread_join`(3THR) or `thr_join`(3THR), `main( )` then waits for the termination of that signal-handling thread, whose ID number is `user_threadID;` after which, `main( )` will `sleep`(3C) for 2 seconds, and then the program terminates.
- The signal-handling thread, `handler`:
  - Assigns the handler `interrupt( )` to handle the signal `SIGINT,` by the call to `sigaction`(2).
  - Resets its own signal set to *not block* the signal, `SIGINT.`
  - Sleeps for 8 seconds to allow time for the user to deliver the signal, `SIGINT,` by pressing the CTRL-C.

```
/* cc thisfile.c -lthread -lpthread */
#define _REENTRANT    /* basic first 3-lines for threads */
#include <pthread.h>
#include <thread.h>

thread_t user_threadID;
sigset_t new;
void *handler( ), interrupt( );

main( int argc, char *argv[ ] ){
   test_argv(argv[1]);

   sigemptyset(&new);
   sigaddset(&new, SIGINT);
   switch(*argv[1])  {

     case '0':   /* POSIX */
       pthread_sigmask(SIG_BLOCK, &new, NULL);
       pthread_create(&user_threadID, NULL, handler, argv[1]);
       pthread_join(user_threadID, NULL);
       break;

     case '1':   /* Solaris */
       thr_sigsetmask(SIG_BLOCK, &new, NULL);
       thr_create(NULL, 0, handler, argv[1], 0, &user_threadID);
       thr_join(user_threadID, NULL, NULL);
       break;
}  /* switch */

   printf("thread handler, # %d, has exited\n",user_threadID);
       sleep(2);
       printf("main thread, # %d is done\n", thr_self( ));
} /* end main */
```

```
struct sigaction act;

void *
handler(char argv1[ ])
 {
        act.sa_handler = interrupt;
        sigaction(SIGINT, &act, NULL);
        switch(*argv1){
          case '0':     /* POSIX */
            pthread_sigmask(SIG_UNBLOCK, &new, NULL);
            break;
          case '1':   /* Solaris */
            thr_sigsetmask(SIG_UNBLOCK, &new, NULL);
            break;
  }
  printf("\n Press CTRL-C to deliver SIGINT signal to the process\n");
  sleep(8);  /* give user time to hit CTRL-C */
}

void
interrupt(int sig)
{
printf("thread %d caught signal %d\n", thr_self( ), sig);
}

void test_argv(char argv1[ ])    {
  if(argv1 == NULL)  {
     printf("use 0 as arg1 to use thr_create( );\n \
     or use 1 as arg1 to use pthread_create( )\n");
     exit(NULL);
  }
}
```

**EXAMPLE 2**

In the last example, the `handler` thread served as a signal-handler while also taking care of activity of its own (in this case, sleeping, although it could have been some other activity). A thread could be completely dedicated to signal-handling simply by waiting for the delivery of a selected signal by blocking with `sigwait`(2). The two subroutines in the previous example, `handler( )` and `interrupt( )`, could have been replaced with the following routine:

```
void *
handler( )
{ int signal;
  printf("thread %d waiting for you to press the CTRL-C keys\n", thr_self( ));
  sigwait(&new, &signal);
  printf("thread %d has received the signal %d \n", thr_self( ), signal);
}
  /*pthread_create( ) and thr_create( ) would use NULL instead of argv[1]
   for the arg passed to handler( ) */
```

In this routine, one thread is dedicated to catching and handling the signal specified by the set `new`, which allows `main()` and all of its other sub-threads, created *after* `pthread_sigmask()` or `thr_sigsetmask()` masked that signal, to continue uninterrupted. Any use of `sigwait`(2) should be such that all threads block the signals passed to `sigwait`(2) at all times. Only the thread that calls `sigwait()` will get the signals. The call to `sigwait`(2) takes two arguments.

For this type of background dedicated signal-handling routine, you may wish to use a Solaris daemon thread by passing the argument `THR_DAEMON` to `thr_create()`.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe and Async-Signal-Safe |

**SEE ALSO**

`sigaction`(2), `sigprocmask`(2), `sigwait`(2), `cond_wait`(3THR), `pthread_create`(3THR), `pthread_join`(3THR), `pthread_self`(3THR), `sigsetops`(3C), `sleep`(3C), `attributes`(5), `standards`(5)

**NOTES**

It is not possible to block signals that cannot be ignored (see `sigaction`(2)). If using the threads library, it is not possible to block the signals `SIGLWP` or `SIGCANCEL,` which are reserved by the threads library. Additionally, it is impossible to unblock the signal `SIGWAITING,` which is always blocked on all threads. This restriction is quietly enforced by the threads library.

Using `sigwait`(2) in a dedicated thread allows asynchronously generated signals to be managed synchronously; however, `sigwait`(2) should never be used to manage synchronously generated signals.

Synchronously generated signals are exceptions that are generated by a thread and are directed at the thread causing the exception. Since `sigwait()` blocks waiting for signals, the blocking thread cannot receive a synchronously generated signal.

If `sigprocmask`(2) is used in a multi-threaded program, it will be the same as if `thr_sigsetmask()` or `pthread_sigmask()` has been called. POSIX leaves the semantics of the call to `sigprocmask`(2) unspecified in a multi-threaded process, so programs that care about POSIX portability should not depend on this semantic.

If a signal is delivered while a thread is waiting on a condition variable, the `cond_wait()` will be interrupted (see `cond_wait`(3THR)) and the handler will be executed. The handler should assume that the lock protecting the condition variable is held.

Signals which are generated synchronously should not be masked. If such a signal is blocked and delivered, the receiving process is killed.

A thread directed `SIGALRM` generated because of a realtime interval timer or process alarm clock is not maskable by a signal masking function, such as thr_sigsetmask(3T), or sigprocmask(2). See `alarm(2)` and `setitimer`(2).

**NAME**

thr_stksegment – get thread stack bottom and stack size

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...[ *library*... ]
#include <thread.h>
#include <sys/signal.h>
int **thr_stksegment**(stack_t*););

**DESCRIPTION**

The stack information provided by `thr_stksegment()` is typically used by debuggers, garbage collectors, and similar applications. Most applications should not require such information. The bottom of the thread stack returned by `thr_stksegment()` points to a part of the stack which may contain data maintained by `libthread`. The user's thread stack starts at a point below the bottom of the stack as returned by `thr_stksegment()`.

**RETURN VALUES**

The `thr_stksegment()` function returns `0` if both the thread stack bottom and stack size were successfully retrieved. Otherwise, it returns a non-zero error code.

**ERRORS**

The `thr_stksegment()` function will fail if:

`EAGAIN` The stack information for the thread is not available because the thread's initialization is not yet complete, or the thread is an internal thread.

The `thr_stksegment()` function may fail if:

`EFAULT` A system call used to get the stack information failed because a bad address was passed to it.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
| --- | --- |
| MT-Level | MT-Safe |

**SEE ALSO**

`thr_create`(3THR), `attributes`(5)

## NAME

thr_suspend, thr_continue – suspend or continue thread execution

## SYNOPSIS

cc -mt [ *flag*... ] *file*...[ *library*... ]

```
#include <thread.h>
int thr_suspend(thread_t target_thread);

int thr_continue(thread_t target_thread);
```

## DESCRIPTION

The `thr_suspend( )` function immediately suspends the execution of the thread specified by *target_thread* . On successful return from `thr_suspend( )` , the suspended thread is no longer executing. Once a thread is suspended, subsequent calls to `thr_suspend( )` have no effect.

The `thr_continue( )` function resumes the execution of a suspended thread. Once a suspended thread is continued, subsequent calls to `thr_continue( )` have no effect.

A suspended thread will not be awakened by a signal. The signal stays pending until the execution of the thread is resumed by `thr_continue( )` .

## RETURN VALUES

If successful, the `thr_suspend( )` and `thr_continue( )` functions return `0` . Otherwise, a non-zero value is returned to indicate the error.

## ERRORS

The `thr_suspend( )` or `thr_continue( )` functions will fail if:

`ESRCH` *target_thread* cannot be found in the current process.

`ECANCELED` *target_thread* was not suspended because a subsequent `thr_continue( )` occurred before the suspend completed.

`EINVAL` When `thr_continue( )` returns `EINVAL`, *target_thread* has died and `thr_join ( )` must be called on it to reclaim its resources.

The `thr_suspend( )` function will fail if:

`EDEADLK` Suspending *target_thread* will cause all threads in the process to be suspended.

## ATTRIBUTES

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

## SEE ALSO

`thr_create`(3THR) , `thr_join`(3THR) , `attributes`(5) , `standards`(5)

**NAME**

thr_yield – yield to another thread

**SYNOPSIS**

cc –mt [ *flag*... ] *file*...[ *library*... ]

#include <thread.h>
void **thr_yield**(void);

**DESCRIPTION**

The thr_yield( ) function causes the current thread to yield its execution in favor of another thread with the same or greater priority.

**RETURN VALUES**

The thr_yield( ) function returns nothing and does not set errno.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe |

**SEE ALSO**

thr_setprio(3THR), 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 **evp*, timer_t **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 willo occur when the timer expires. 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, not bound |
| 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 LWP 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 effective user of the calling LWP is not superuser.

**ATTRIBUTES**

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

| ATTRIBUTE TYPE | ATTRIBUTE VALUE |
|---|---|
| MT-Level | MT-Safe with exceptions |

**SEE ALSO**

`exec`(2), `fork`(2), `time`(2), `clock_settime`(3RT), `signal`(3C), `timer_delete`(3RT), `timer_settime`(3RT), `attributes`(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 |
| --- | --- |
| MT-Level | MT-Safe with exceptions |

**SEE ALSO**

`timer_create`(3RT), `attributes`(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 **value*, struct itimerspec **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 or LWP 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 |
|---|---|
| MT-Level | Async-Signal-Safe |

**SEE ALSO**

clock_settime(3RT) , timer_create(3RT) , timer_delete(3RT) , attributes(5) , time(3HEAD)

# Index


## A

access dynamic thread scheduling
  – thr_getprio 310
  – thr_setprio 310
access dynamic thread scheduling parameters
  – pthread_getschedparam 160
  – pthread_setschedparam 160
aio_cancel — cancel asynchronous I/O request 22
aio_fsync — asynchronous file synchronization 26
aio_read — asynchronous read and write operations 31
aio_return — retrieve return status of asynchronous I/O operation 34
aio_suspend — wait for asynchronous I/O request 35
aio_write — asynchronous write to a file 39
aiocancel — cancel an asynchronous operation 21
aioread – read or write asynchronous I/O operations 28
aiowait — wait for completion of asynchronous I/O operation 37
aiowrite – read or write asynchronous I/O operations 28
allocate and deallocate process handles for libthread_db
  – td_ta_delete 272
  – td_ta_get_ph 272
  – td_ta_new 272
asynchronous file synchronization
  — aio_sync 26
asynchronous I/O
  — aio_cancel 22
  — aiocancel 21
  — aiowait 37
  retrieve return status — aio_return 34
asynchronous read and write operations
  — aio_read, aio_write 31
asynchronous write to a file — aio_write 39

## B

bind or unbind the current thread with the door server pool
  – door_bind 56
  – door_unbind 56

## C

cancellation — overview of concepts related to POSIX thread cancellation 42
  Cancel-Safe 45
  Cancellation 42
  Cancellation Points 43
  Cancellation State 44
  Cancellation Type 44
  Cleanup Handlers 44
  Planning Steps 42
  POSIX Threads Only 45
change or examine calling thread's signal mask — pthread_sigmask 207, 322
change the priority ceiling of a mutex

– pthread_mutex_getprioceiling 184
– pthread_mutex_setprioceiling 184
clock_getres – high-resolution clock operations 48
clock_gettime – high-resolution clock operations 48
clock_settime – high-resolution clock operations 48
collect target process statistics for libthread_db
– td_ta_enable_stats 262
– td_ta_get_stats 262
– td_ta_reset_stats 262
compare thread IDs — pthread_equal 155
concepts related to condition variables — condition 54
concepts relating to mutual exclusion locks — mutex 98
cond_broadcast – condition variables 50
cond_destroy – condition variables 50
cond_init – condition variables 50
Condition Signaling 51
Condition Wait 51
Destroy 52
Initialize 50
cond_signal – condition variables 50
cond_timedwait – condition variables 50
cond_wait – condition variables 50
condition — concepts related to condition variables 54
Condition Signaling 55
Condition Wait 54
Destroy 55
Initialize 54
condition variables
– cond_broadcast 50
– cond_destroy 50
– cond_init 50
– cond_signal 50
– cond_timedwait 50
– cond_wait 50
convert a thread id or thread address to a thread handle
– td_ta_map_addr2thr 271
– td_ta_map_id2thr 271
create a door descriptor — door_create 62
create a thread — pthread_create 150
create a tread — thr_create 292
create cancellation point in the calling thread. — pthread_testcancel 212
create thread-specific data key — pthread_key_create 165

**D**

delete thread-specific data key — pthread_key_delete 167
detach a thread — pthread_detach 154
door_bind – bind or unbind the current thread with the door server pool 56
door_call — invoke the function associated with a door descriptor 59
door_create — create a door descriptor 62
door_cred — return credential information associated with the client 64
door_info — return information associated with a door descriptor 65
door_return — return from a door invocation 67
door_revoke — revoke access to a door descriptor 68
door_server_create — specify an alternative door server thread creation function 69
door_unbind – bind or unbind the current thread with the door server pool 56

**E**

enable or disable cancellation — pthread_setcancelstate 203
enabling or disabling cancellation — pthread_setcancelstate 203

**F**

fdatasync — synchronize a file’s data 72

**G**

get a synchronization object handle from a synchronization object’s address — td_ta_map_addr2sync 270

get a thread’s thread-specific data for libthread_db library of interfaces — td_thr_tsd 290
get and set prioceiling attribute of mutex attribute object
– pthread_mutexattr_getprioceiling 169
– pthread_mutexattr_setprioceiling 169
get and set process-shared attribute
– pthread_mutexattr_getpshared 174
– pthread_mutexattr_setpshared 174
get and set process-shared attribute of read-write lock attributes object
– pthread_rwlockattr_getpshared 193
– pthread_rwlockattr_setpshared 193
get and set protocol attribute of mutex attribute object
– pthread_mutexattr_getprotocol 171
– pthread_mutexattr_setprotocol 171
get calling thread’s ID — pthread_self 202, 321
get execution time limits — sched_rr_get_interval 222
get message queue attributes — mq_getattr 85
get or set a mutex type
– pthread_mutexattr_gettype 178
– pthread_mutexattr_settype 178
get or set contentionscope attribute
– pthread_attr_getscope 131
– pthread_attr_setscope 131
get or set detachstate attribute
– pthread_attr_getdetachstate 124
– pthread_attr_setdetachstate 124
get or set inheritsched attribute
– pthread_attr_getinheritsched 127
– pthread_attr_setinheritsched 127
get or set level of concurrency
– pthread_getconcurrency 158
– pthread_setconcurrency 158
get or set schedparam attribute
– pthread_attr_getschedparam 129
– pthread_attr_setschedparam 129
get or set schedpolicy attribute
– pthread_attr_getschedpolicy 130
– pthread_attr_setschedpolicy 130
get or set stackaddr attribute
– pthread_attr_getstackaddr 133
– pthread_attr_setstackaddr 133
get or set stacksize attribute
– pthread_attr_getstacksize 134
– pthread_attr_setstacksize 134
get or set the process-shared condition variable attributes
– pthread_condattr_getpshared 140
– pthread_condattr_setpshared 140
get or set the thread guardsize attribute
– pthread_attr_getguardsize 125
– pthread_attr_setguardsize 125
get scheduling parameter limits
– sched_get_priority_max 220
– sched_get_priority_min 220
get scheduling parameters — sched_getparam 219
get scheduling policy — sched_getscheduler 221
get thread information in libthread_db library of interfaces — td_thr_get_info 281
gets the total number of threads in a process for libthread_db — td_ta_get_nthreads 269

# I

I/O, asynchronous
cancel request — aio_cancel 22
file synchronization — aio_sync 26
retrieve return status — aio_return 34
I/O, requests
list — lio_listio 80
initialization function for libthread_db library of interfaces — td_init 256
initialize and destroy mutex attributes object
– pthread_mutexattr_destroy 181
– pthread_mutexattr_init 181
initialize and destroy read-write lock attributes object
– pthread_rwlockattr_destroy 194
– pthread_rwlockattr_init 194
initialize and destroy threads attribute object
– pthread_attr_destroy 135
– pthread_attr_init 135
initialize dynamic package — pthread_once 192
initialize or destroy a mutex

– pthread_mutex_destroy 186
– pthread_mutex_init 186
initialize or destroy a read-write lock object
– pthread_rwlock_destroy 195
– pthread_rwlock_init 195
initialize or destroy condition variable attributes object
– pthread_condattr_destroy 142
– pthread_condattr_init 142
initialize or destroy condition variables
– pthread_cond_destroy 144
– pthread_cond_init 144
interfaces in libthread_db that target process memory access
– ps_pdread 119
– ps_pdwrite 119
– ps_ptread 119
– ps_ptwrite 119
invoke the function associated with a door descriptor — door_call 59
iterate over the set of locks owned by a thread — td_thr_lockowner 285
iterator functions on process handles from libthread_db library of interfaces
– td_ta_sync_iter 275
– td_ta_thr_iter 275
– td_ta_tsd_iter 275

## L

library of interfaces for monitoring and manipulating threads-related aspects of multithreaded programs — libthread_db 75
libthread_db — library of interfaces for monitoring and manipulating threads-related aspects of multithreaded programs 75
lio_listio — list directed I/O 80
list directed I/O — lio_listio 80
lock or attempt to lock a read-write lock object for reading
– pthread_rwlock_rdlock 197
– pthread_rwlock_tryrdlock 197
lock or attempt to lock a read-write lock object for writing
– pthread_rwlock_trywrlock 200
– pthread_rwlock_wrlock 200
lock or unlock a mutex
– pthread_mutex_lock 188
– pthread_mutex_trylock 188
– pthread_mutex_unlock 188
looks up the symbol in the symbol table of the load object in the target process
– ps_pglobal_lookup 118

## M

make a mutex consistent after owner death — pthread_mutex_consistent_np 182
manage thread signals for libthread_db
– td_thr_setsigpending 287
– td_thr_sigsetmask 287
manage thread-specific data
– pthread_getspecific 162
– pthread_setspecific 162
memory object, shared
open — shm_open 248
remove — shm_unlink 251
message queue
close — mq_close 84
notify process (or thread) — mq_notify 86
open — mq_open 88
receive a message from — mq_receive 92
remove — mq_unlink 97
send message to — mq_send 94
set attributes — mq_setattr 96
mq_close — close a message queue 84
mq_getattr — get message queue attributes 85
mq_notify — notify process (or thread) that a message is available on a queue 86
mq_open — open a message queue 88
mq_receive — receive a message from a message queue 92
mq_send — send a message to a message queue 94
mq_setattr — set/get message queue attributes 96
mq_unlink — remove a message queue 97
mutex — concepts relating to mutual exclusion locks 98
Caveats 99

Initialization 98
mutex_destroy – mutual exclusion locks 100
mutex_init – mutual exclusion locks 100
Destroy 103
Dynamically Allocated Mutexes 110
Initialize 100
Interprocess Locking 106
Lock and Unlock 102
Multiple Instruction Single Data 105
Single Gate 105
Solaris Interprocess Robust Locking 108
mutex_lock – mutual exclusion locks 100
mutex_trylock – mutual exclusion locks 100
mutex_unlock – mutual exclusion locks 100
mutual exclusion locks
– mutex_destroy 100
– mutex_init 100
– mutex_lock 100
– mutex_trylock 100
– mutex_unlock 100

## N

nanosleep — high resolution sleep 112

## O

operations on a synchronization object in libthread_db
– td_sync_get_info 258
– td_sync_setstate 258
– td_sync_waiters 258
overview of concepts related to POSIX thread cancellation — cancellation 42

## P

placeholder for future logging functionality — td_log 257
pop a thread cancellation cleanup handler — pthread_cleanup_pop 138
preemption control
– schedctl_exit 217
– schedctl_init 217
– schedctl_lookup 217
– schedctl_start 217
– schedctl_stop 217
proc_service — process service interfaces 114
IA 113
SPARC 113
process and LWP control in libthread_db
– ps_kill 120
– ps_lcontinue 120
– ps_lrolltoaddr 120
– ps_lstop 120
– ps_pcontinue 120
– ps_pstop 120
process service interfaces — proc_service 114
ps_kill – process and LWP control in libthread_db 120
ps_lcontinue – process and LWP control in libthread_db 120
ps_lgetfpregs – routines that access the target process register in libthread_db 116
ps_lgetregs – routines that access the target process register in libthread_db 116
ps_lgetxregs – routines that access the target process register in libthread_db 116
ps_lgetxregsize – routines that access the target process register in libthread_db 116
ps_lrolltoaddr – process and LWP control in libthread_db 120
ps_lsetfpregs – routines that access the target process register in libthread_db 116
ps_lsetregs – routines that access the target process register in libthread_db 116
ps_lsetxregs – routines that access the target process register in libthread_db 116
ps_lstop – process and LWP control in libthread_db 120
ps_pcontinue – process and LWP control in libthread_db 120
ps_pdread – interfaces in libthread_db that target process memory access 119

ps_pdwrite – interfaces in libthread_db that target process memory access 119
ps_pglobal_lookup – look up a symbol in the symbol table of the load object in the target process 118
ps_pglobal_sym – look up a symbol in the symbol table of the load object in the target process 118
ps_pstop – process and LWP control in libthread_db 120
ps_ptread – interfaces in libthread_db that target process memory access 119
ps_ptwrite – interfaces in libthread_db that target process memory access 119
pthread_atfork — register fork handlers 122
pthread_attr_destroy – initialize and destroy threads attribute object 135
pthread_attr_getdetachstate – get or set detachstate attribute 124
pthread_attr_getguardsize – get or set the thread guardsize attribute 125
pthread_attr_getinheritsched – get or set inheritsched attribute 127
pthread_attr_getschedparam – get or set schedparam attribute 129
pthread_attr_getschedpolicy – get or set schedpolicy attribute 130
pthread_attr_getscope – get or set contentionscope attribute 131
pthread_attr_getstackaddr – get or set stackaddr attribute 133
pthread_attr_getstacksize – get or set stacksize attribute 134
pthread_attr_init – initialize and destroy threads attribute object 135
pthread_attr_setdetachstate – get or set detachstate attribute 124
pthread_attr_setguardsize – get or set the thread guardsize attribute 125
pthread_attr_setinheritsched – get or set inheritsched attribute 127
pthread_attr_setschedparam – get or set schedparam attribute 129
pthread_attr_setschedpolicy – get or set schedpolicy attribute 130
pthread_attr_setscope – get or set contentionscope attribute 131
pthread_attr_setstackaddr – get or set stackaddr attribute 133
pthread_attr_setstacksize – get or set stacksize attribute 134
pthread_cleanup_pop — pop a thread cancellation cleanup handler 138
pthread_cleanup_push — push a thread cancellation cleanup handler 139
pthread_cond_broadcast – signal or broadcast a condition 146
pthread_cond_destroy – initialize or destroy condition variables 144
pthread_cond_init – initialize or destroy condition variables 144
pthread_cond_signal – signal or broadcast a condition 146
pthread_cond_timedwait – wait on a condition 148
pthread_cond_wait – wait on a condition 148
pthread_condattr_destroy – initialize or destroy condition variable attributes object 142
pthread_condattr_getpshared – get or set the process-shared condition variable attributes 140
pthread_condattr_init – initialize or destroy condition variable attributes object 142
pthread_condattr_setpshared – get or set the process-shared condition variable attributes 140
pthread_create — create a thread 150
pthread_detach — detach a thread 154
pthread_equal — compare thread IDs 155
pthread_exit — terminate calling thread 156
pthread_getconcurrency – get or set level of concurrency 158
pthread_getschedparam – access dynamic thread scheduling parameters 160

pthread_getspecific – manage thread-specific data 162
pthread_join — wait for thread termination 163
pthread_key_create — create thread-specific data key 165
pthread_key_delete — delete thread-specific data key 167
pthread_mutex_consistent_np — make a mutex consistent after owner death 182
pthread_mutex_destroy – initialize or destroy a mutex 186
pthread_mutex_getprioceiling – change the priority ceiling of a mutex 184
pthread_mutex_init – initialize or destroy a mutex 186
pthread_mutex_lock – lock or unlock a mutex 188
pthread_mutex_setprioceiling – change the priority ceiling of a mutex 184
pthread_mutex_trylock – lock or unlock a mutex 188
pthread_mutex_unlock – lock or unlock a mutex 188
pthread_mutexattr_destroy – initialize and destroy mutex attributes object 181
pthread_mutexattr_getprioceiling – get and set prioceiling attribute of mutex attribute object 169
pthread_mutexattr_getprotocol – get and set protocol attribute of mutex attribute object 171
pthread_mutexattr_getpshared – get and set process-shared attribute 174
pthread_mutexattr_gettype – get or set a mutex type 178
pthread_mutexattr_init – initialize and destroy mutex attributes object 181
pthread_mutexattr_setprioceiling – get and set prioceiling attribute of mutex attribute object 169
pthread_mutexattr_setprotocol – get and set protocol attribute of mutex attribute object 171
pthread_mutexattr_setpshared – get and set process-shared attribute 174
pthread_mutexattr_settype – get or set a mutex type 178
pthread_once — initialize dynamic package 192
pthread_rwlock_destroy – initialize or destroy a read-write lock object 195
pthread_rwlock_init – initialize or destroy a read-write lock object 195
pthread_rwlock_rdlock – lock or attempt to lock a read-write lock object for reading 197
pthread_rwlock_tryrdlock – lock or attempt to lock a read-write lock object for reading 197
pthread_rwlock_trywrlock – lock or attempt to lock a read-write lock object for writing 200
pthread_rwlock_unlock — unlock a read-write lock object 199
pthread_rwlock_wrlock – lock or attempt to lock a read-write lock object for writing 200
pthread_rwlockattr_destroy – initialize and destroy read-write lock attributes object 194
pthread_rwlockattr_getpshared – get and set process-shared attribute of read-write lock attributes object 193
pthread_rwlockattr_init – initialize and destroy read-write lock attributes object 194
pthread_rwlockattr_setpshared – get and set process-shared attribute of read-write lock attributes object 193
pthread_self — get calling thread's ID 202
pthread_setcancelstate — enable or disable cancellation 203
pthread_setcanceltype — set the cancellation type of a thread 205

pthread_setconcurrency – get or set level of concurrency 158
pthread_setschedparam – access dynamic thread scheduling parameters 160
pthread_setspecific – manage thread-specific data 162
pthread_sigmask — change or examine calling thread’s signal mask 207
pthread_testcancel — create cancellation point in the calling thread. 212
push a thread cancellation cleanup handler — pthread_cleanup_push 139

## R

read or write asynchronous I/O operations
– aioread 28
– aiowrite 28
reading and writing thread registers in libthread_db
– td_thr_getfpregs 279
– td_thr_getgregs 279
– td_thr_getxregs 279
– td_thr_getxregsize 279
– td_thr_setfpregs 279
– td_thr_setgregs 279
– td_thr_setxregs 279
register fork handlers — pthread_atfork 122
return credential information associated with the client — door_cred 64
return from a door invocation — door_return 67
return information associated with a door descriptor — door_info 65
return the synchronization handle for the object on which a thread is blocked — td_thr_sleepinfo 289
revoke access to a door descriptor — door_revoke 68
routines that access the target process register in libthread_db
– ps_lgetfpregs 116
– ps_lgetregs 116
– ps_lgetxregs 116
– ps_lgetxregsize 116
– ps_lsetfpregs 116
– ps_lsetregs 116
– ps_lsetxregs 116
rw_rdlock() – acquire a read lock 214
rw_tryrdlock() – acquire a read lock 214
rw_trywrlock() – acquire a write lock 214
rw_unlock() – unlock a readers/writer lock 214
rw_wrlock() – acquire a write lock 214
rwlock_destroy() – destroy a readers/writer lock 214
rwlock_init() – initialize a readers/writer lock 214

## S

sched_get_priority_max – get scheduling parameter limits 220
sched_get_priority_min – get scheduling parameter limits 220
sched_getparam — get scheduling parameters 219, 223
sched_getscheduler — get scheduling policy 221
sched_rr_get_interval — get execution time limits 222
sched_setparam — set/get scheduling parameters 223
sched_setscheduler — set scheduling policy and scheduling parameters 226
sched_yield — yield processor 229
schedctl_exit – preemption control 217
schedctl_init – preemption control 217
schedctl_lookup – preemption control 217
schedctl_start – preemption control 217
schedctl_stop – preemption control 217
sem_close — close a named semaphore 234
sem_destroy — destroy an unnamed semaphore 235
sem_getvalue — get the value of a semaphore 236
sem_init — initialize an unnamed semaphore 237
sem_open — initialize/open a named semaphore 239
sem_post — increment the count of a semaphore 242

sem_trywait – acquire or wait for a
semaphore 245
sem_unlink — remove a named
semaphore 244
sem_wait – acquire or wait for a
semaphore 245
sema_destroy() – destroy a semaphore 230
sema_init() – initialize a semaphore 230
sema_post() – increment a semaphore 230
sema_trywait() – decrement a semaphore 230
sema_wait() – decrement a semaphore 230
semaphore
acquire or wait for – sem_wait,
sem_trywait 245
close a named one — sem_close 234
destroy an unnamed one —
sem_destroy 235
get the value — sem_getvalue 236
increment the count — sem_post 242
initialize an unnamed one — sem_init 237
initialize / open a named one —
sem_open 239
remove a named one — sem_unlink 244
set concurrency level for target process —
td_ta_setconcurrency 274
set scheduling policy and scheduling
parameters —
sched_setscheduler 226
set the cancellation type of a thread —
pthread_setcanceltype 205
set the priority of a thread —
td_thr_setprio 286
set / get scheduling parameters
— sched_getparam 223
— sched_setparam 223
shared memory object
open — shm_open 248
remove — shm_unlink 251
shm_open — open a shared memory
object 248
shm_unlink — remove a shared memory
object 251
signal
queue one to a process — sigqueue 252
wait for queued signals – sigwaitinfo,
sigtimedwait 254
signal or broadcast a condition
– pthread_cond_broadcast 146
– pthread_cond_signal 146
sigqueue — queue a signal to a process 252
sigtimedwait – wait for queued signals 254
sigwaitinfo – wait for queued signals 254
sleep
high resolution — nanosleep 112
specify an alternative door server thread
creation function —
door_server_create 69
suspend and resume threads in libthread_db
– td_thr_dbresume 277
– td_thr_dbsuspend 277
synchronize a file's data
— fdatasync 72


## T


td_event_addset – thread events in
libthread_db 264
td_event_delset – thread events in
libthread_db 264
td_event_emptyset – thread events in
libthread_db 264
td_event_fillset – thread events in
libthread_db 264
td_eventisempty – thread events in
libthread_db 264
td_eventismember – thread events in
libthread_db 264
td_init — initialization function for libthread_db
library of interfaces 256
td_log — placeholder for future logging
functionality 257
td_sync_get_info – operations on a
synchronization object in
libthread_db 258
td_sync_setstate – operations on a
synchronization object
in libthread_db 258
td_sync_waiters – operations on a
synchronization object
in libthread_db 258
td_ta_delete – allocate and deallocate process
handles for libthread_db 272
td_ta_enable_stats – collect target process
statistics for libthread_db 262

td_ta_event_addr – thread events in libthread_db 264
Event Set Manipulation Macros 267
td_ta_event_getmsg – thread events in libthread_db 264
td_ta_get_nthreads — gets the total number of threads in a process for libthread_db 269
td_ta_get_ph – allocate and deallocate process handles for libthread_db 272
td_ta_get_stats – collect target process statistics for libthread_db 262
td_ta_map_addr2sync — get a synchronization object handle from a synchronization object’s address 270
td_ta_map_addr2thr – convert a thread id or thread address to a thread handle 271
td_ta_map_id2thr – convert a thread id or thread address to a thread handle 271
td_ta_new – allocate and deallocate process handles for libthread_db 272
td_ta_reset_stats – collect target process statistics for libthread_db 262
td_ta_set_event – thread events in libthread_db 264
td_ta_setconcurrency — set concurrency level for target process 274
td_ta_sync_iter – iterator functions on process handles from libthread_db library of interfaces 275
td_ta_thr_iter – iterator functions on process handles from libthread_db library of interfaces 275
td_ta_tsd_iter – iterator functions on process handles from libthread_db library of interfaces 275
td_thr_clear_event – thread events in libthread_db 264
td_thr_dbresume – suspend and resume threads in libthread_db 277
td_thr_dbsuspend – suspend and resume threads in libthread_db 277
td_thr_event_enable – thread events in libthread_db 264
td_thr_event_getmsg – thread events in libthread_db 264
td_thr_get_info — get thread information in libthread_db library of interfaces 281
td_thr_getfpregs – reading and writing thread registers in libthread_db 279
td_thr_getgregs – reading and writing thread registers in libthread_db 279
Intel IA 280
SPARC 280
td_thr_getxregs – reading and writing thread registers in libthread_db 279
td_thr_getxregsize – reading and writing thread registers in libthread_db 279
td_thr_lockowner — iterate over the set of locks owned by a thread 285
td_thr_set_event – thread events in libthread_db 264
td_thr_setfpregs – reading and writing thread registers in libthread_db 279
td_thr_setgregs – reading and writing thread registers in libthread_db 279
td_thr_setprio — set the priority of a thread 286
td_thr_setsigpending – manage thread signals for libthread_db 287
td_thr_setxregs – reading and writing thread registers in libthread_db 279
td_thr_sigsetmask – manage thread signals for libthread_db 287
td_thr_sleepinfo — return the synchronization handle for the object on which a thread is blocked 289
td_thr_tsd — get a thread’s thread-specific data for libthread_db library of interfaces 290
td_thr_validate — test a thread handle for validity 291
tda_ta_clear_event – thread events in libthread_db 264
terminate calling thread — pthread_exit 156
terminate the calling thread — thr_exit 307
test a thread handle for validity — td_thr_validate 291
thr_continue – continue thread execution 328
thr_create — create a tread 292

thr_exit — terminate the calling thread 307
thr_getconcurrency – get thread concurrency level 309
thr_getprio – access dynamic thread scheduling 310
Contentionscope 310
Policy 310
Priority 310
Scheduling 310
thr_getspecific – thread-specific-data functions 314
thr_join — wait for thread termination 312
thr_keycreate – thread-specific-data functions 314
Create Key 314
Get Value 314
Set Value 314
thr_main — identifies the calling thread as the main thread or not the main thread 318
thr_self — get calling thread's ID 321
thr_setconcurrency – set thread concurrency level 309
thr_setprio – access dynamic thread scheduling 310
thr_setspecific – thread-specific-data functions 314
thr_sigsetmask — change or examine calling thread's signal mask 322
thr_stksegment — get thread stack bottom and size 327
thr_suspend – suspend thread execution 328
thr_yield — thread yield to another thread 329
thread events in libthread_db
– td_event_addset 264
– td_event_delset 264
– td_event_emptyset 264
– td_event_fillset 264
– td_eventisempty 264
– td_eventismember 264
– td_ta_event_addr 264
– td_ta_event_getmsg 264
– td_ta_set_event 264
– td_thr_clear_event 264
– td_thr_event_enable 264
– td_thr_event_getmsg 264
– td_thr_set_event 264
– tda_ta_clear_event 264
thread yield to another thread — thr_yield 329
thread-specific-data functions
– thr_getspecific 314
– thr_keycreate 314
– thr_setspecific 314
timer_getoverrun – per-process timers 333
timer_gettime – per-process timers 333
timer_settime – per-process timers 333

## U

unlock a read-write lock object — pthread_rwlock_unlock 199

## W

wait for thread termination — pthread_join 163, 312
wait on a condition
– pthread_cond_timedwait 148
– pthread_cond_wait 148

## Y

yield processor — sched_yield 229