Programming With Threads 2

The Threads Library

User-level multithreading is implemented through the threads library, `libthread` (see section 3T in the *man Pages(3): Library Routines*). The threads library supports signals, schedules runable entities, and handles multiple tasks simultaneously.

This chapter discusses some of the general `libthread` routines, starting with the basic ways to create threads and becoming more advanced.

Create a Thread—the Basics	*thr_create(3T)*	*page 12*
Get the Thread Identifier	*thr_self(3T)*	*page 14*
Yield Thread Execution	*thr_yield(3T)*	*page 14*
Suspend or Continue Thread Execution	*thr_suspend(3T)*	*page 15*
	thr_continue(3T)	*page 15*
Send a Signal to a Thread	*thr_kill(3T)*	*page 16*
Access the Signal Mask of the Calling Thread	*thr_sigsetmask(3T)*	*page 16*
Terminate a Thread	*thr_exit(3T)*	*page 17*
Wait for Thread Termination	*thr_join(3T)*	*page 19*
Maintain Thread-Specific Data	*thr_keycreate(3T)*	*page 23*
	thr_setspecific(3T)	*page 24*
	thr_getspecific(3T)	*page 24*
Create a Thread—Advanced Features	*thr_create(3T)*	*page 28*

2

Get the Minimal Stack Size	*thr_min_stack(3T)*	*page 32*
Get and Set Thread Concurrency Level	*thr_getconcurrency(3T)*	*page 33*
	thr_setconcurrency(3T)	*page 34*
Get and Set Thread Priority	*thr_getprio(3T)*	*page 35*
	thr_setprio(3T)	*page 35*

Create a Thread—the Basics

The `thr_create`(3T) routine is the most elaborate of all the threads library routines. The explanations in this section are for those cases when you can use the default values for the `thr_create`() arguments.

More advanced `thr_create`() use, including explanations of nondefault argument values, is covered toward the end of this chapter in "Create a Thread—Advanced Features" on page 28.

`thr_create`*(3T)*

Use `thr_create`() to add a new thread of control to the current process. Note that the new thread does not inherit pending signals, but it does inherit priority and signal masks.

```
#include <thread.h>

int thr_create(void *stack_base, size_t stack_size,
    void *(*start_routine) (void *), void *arg, long flags,
    thread_t *new_thread);

size_t thr_min_stack(void);
```

stack_base—Contains the address for the stack that the new thread uses. If *stack_base* is `NULL` then `thr_create`() allocates a stack for the new thread with at least *stack_size* bytes.

stack_size—Contains the size, in number of bytes, for the stack that the new thread uses. If *stack_size* is zero, a default size is used. In most cases, a zero value works best.

There is no general need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of `mmap`(2) to make the allocations.)

start_routine—Contains the function with which the new thread begins execution. If *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine* (see `thr_exit`(3T)).

flags—Specifies attributes for the created thread. In most cases a zero value works best.

The value in *flags* is constructed from the bitwise inclusive OR of the following. (The last four flags are explained more fully in "Create a Thread—Advanced Features" on page 28.)

`THR_DETACHED` Detaches the new thread so that its thread ID and other resources can be reused as soon as the thread terminates. Set this when you do not want to wait for the thread to terminate.

When there is no explicit synchronization to prevent it, an unsuspended, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `thr_create`().

`THR_SUSPENDED` Suspends the new thread and does not execute *start_routine* until the thread is started by `thr_continue`().

`THR_BOUND` Permanently binds the new thread to an LWP (the new thread is a *bound thread*).

`THR_NEW_LWP` Increases the concurrency level for unbound threads by one.

`THR_DAEMON` Marks the new thread as a daemon.

new_thread—Points to a location (when *new_thread* is not NULL) where the ID of the new thread is stored. In most cases a zero value works best.

Return Values—`thr_create`() returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_create`() fails and returns the corresponding value.

EAGAIN A system limit is exceeded, such as when too many LWPs have been created.

ENOMEM Not enough memory was available to create the new thread.

EINVAL *stack_base* is not NULL and *stack_size* is less than the value returned by `thr_min_stack`().

Get the Thread Identifier

`thr_self`*(3T)*

`Use thr_self`(3T) to get the ID of the calling thread.

```
#include <thread.h>

thread_t thr_self(void)
```

Return Values—`thr_self`() returns the ID of the calling thread.

Yield Thread Execution

`thr_yield`*(3T)*

`thr_yield`() causes the current thread to yield its execution in favor of another thread with the same or greater priority.

```
#include <thread.h>

void thr_yield(void);
```

Suspend or Continue Thread Execution

`thr_suspend`*(3T)*

`thr_suspend`() suspends thread execution.

```
#include <thread.h>

int thr_suspend(thread_t target_thread);
```

`thr_suspend`() 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.

Return Values—`thr_suspend()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_suspend`() fails and returns the corresponding value.

ESRCH *target_thread* cannot be found in the current process.

`thr_continue`*(3T)*

`thr_continue`() resumes the execution of a suspended thread. Once a suspended thread is continued, subsequent calls to `thr_continue`() have no effect.

```
#include <thread.h>

int thr_continue(thread_t target_thread);
```

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—`thr_continue()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_continue`() fails and returns the corresponding value.

ESRCH *target_thread* cannot be found in the current process.

Send a Signal to a Thread

thr_kill(3T)

thr_kill() sends a signal to a thread.

```
#include <thread.h>
#include <signal.h>

int thr_kill(thread_t target_thread, int sig);
```

thr_kill() sends the signal *sig* to the thread specified by *target_thread*. *target_thread* must be a thread within the same process as the calling thread. The *sig* argument must be from the list given in signal(5).

When *sig* is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of *target_thread*.

Return Values—thr_kill() returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, thr_kill() fails and returns the corresponding value.

EINVAL *sig* is not a valid signal number.

ESRCH *target_thread* cannot be found in the current process.

Access the Signal Mask of the Calling Thread

thr_sigsetmask(3T)

Use thr_sigsetmask() to change or examine the signal mask of the calling thread.

```
#include <thread.h>
#include <signal.h>

int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset);
```

how determines how the signal set is changed. It can have one of the following values:

`SIG_BLOCK`—Add *set* to the current signal mask, where *set* indicates the set of signals to block.

`SIG_UNBLOCK`—Delete *set* from the current signal mask, where *set* indicates the set of signals to unblock.

`SIG_SETMASK`—Replace the current signal mask with *set*, where *set* indicates the new signal mask.

When the value of *set* is NULL, the value of *how* is not significant and the signal mask of the thread is unchanged. So, to inquire about currently blocked signals, assign a NULL value to the *set* argument.

oset points to the space where the previous signal mask is stored, unless it is NULL.

Return Values—`thr_sigsetmask`() returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_sigsetmask`() fails and returns the corresponding value.

EINVAL The value of *set* is not NULL and the value of *how* is not defined.

EFAULT Either *set* or *oset* is not a valid address.

Terminate a Thread

`thr_exit`*(3T)*

Use `thr_exit`() to terminate a thread.

```
#include <thread.h>

void thr_exit(void *status);
```

The `thr_exit`() function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by *status* are retained until the thread is waited for. Otherwise, *status* is ignored and the thread's ID can be reclaimed immediately.

Return Values—When the calling thread is the last nondaemon thread in the process, the process terminates with a status of zero. When the initial thread returns from `main`() the process exits with a status equal to the return value.

Finishing Up

A thread can terminate its execution in two ways. The first is simply to return from its first (outermost) procedure. The second is to call `thr_exit`(), supplying an exit code. What happens next depends upon how the *flags* parameter was set when the thread was created.

The default behavior of a thread (which happens when the appropriate bit in the *flags* parameter is left as zero) is to remain until some other thread has acknowledged its demise by "joining" with it. The result of the join is that the joining thread picks up the exit code of the dying thread and the dying thread vanishes. You can set a bit in the *flags* parameter, by ORing into it `THR_DETACHED`, to make the thread disappear immediately after it calls `thr_exit`() or after it returns from its first procedure. In this case, the exit code is not picked up by any thread.

An important special case arises when the main thread, the one that existed initially, returns from the main procedure or calls `exit`(). This action causes the entire process to be terminated, along with all its threads. So take care to ensure that the main thread does not return from `main` prematurely.

Note that when the main thread merely calls `thr_exit`(), it terminates only itself—the other threads in the process, as well as the process, continue to exist. (The process terminates when all threads terminate.)

Note also that if a thread is non-detached, then it is very important that some thread join with it after it terminates—otherwise the resources of that thread are not released for use by new threads. So when you do not want a thread to be joined, create it as a detached thread.

An additional *flags* argument to `thr_create`() is `THR_DAEMON`. Threads created with this flag, daemon threads, are automatically terminated when all nondaemon threads have terminated. These daemon threads are especially useful within libraries.

Daemon threads can be created within library routines—as daemon threads they are effectively invisible to the rest of the program. When all other threads in the program (the threads you were aware of creating) terminate, these daemon threads automatically terminate. If they were not daemon threads, they would not terminate when the other threads do, and the process would not exit.

Wait for Thread Termination

`thr_join`*(3T)*

Use the `thr_join`() function to wait for a thread to terminate.

```
#include <thread.h>

int thr_join(thread_t wait_for, thread_t *departed,
   void **status);
```

The `thr_join`() function blocks the calling thread until the thread specified by *wait_for* terminates. The specified thread must be in the current process and must not be detached. When *wait_for* is `(thread_t)0`, then `thr_join`() waits for any undetached thread in the process to terminate. In other words, when no thread identifier is specified, any undetached thread that exits causes `thr_join`() to return.

When *departed* is not NULL, it points to a location that is set to the ID of the terminated thread when `thr_join`() returns successfully. When *status* is not NULL, it points to a location that is set to the exit status of the terminated thread when `thr_join`() returns successfully.

When a stack was specified when the thread was created, the stack can be reclaimed when `thr_join`() returns. The thread identifier returned by a successful `thr_join`() can then be used by `thr_create`().

Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of ESRCH.

Return Values—`thr_join`() returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_join`() fails and returns the corresponding value.

ESRCH *wait_for* is not a valid, undetached thread in the current process.

EDEADLK *wait_for* specifies the calling thread.

Final Steps

The `thr_join`() routine takes three arguments, giving you some flexibility in its use. When you want the caller to wait until a specific (nondetached) thread terminates, supply that thread's ID as the first argument. When you want the caller to wait until any nondetached thread terminates, supply a zero for the first argument.

When the caller wants to find out who the terminated thread is, the second argument should be the address of storage into which the defunct thread's ID will be placed.

Otherwise, supply a zero for this argument. Finally, if you are interested in the exit code of the defunct thread, supply the address of an area to receive it.

A thread can wait until all nondaemon threads have terminated by executing the following:

```
while(thr_join(0, 0, 0) == 0)
   ;
```

The declaration for the third parameter of `thr_join`(), `void` **, might look strange. The corresponding argument of `thr_exit`() is *void* *. The intent is that you pass an arbitrary 4-byte item as the exit code. The C for "arbitrary 4-byte argument" cannot be `void`, because that means that there is no argument. So it is `void` *. Because the third parameter of `thr_join`() is an output parameter that must point to whatever is supplied by `thr_exit()`, its type is necessarily `void` **.

Remember that `thr_join()` works only for target threads that are nondetached. When there is no reason to synchronize with the termination of a particular thread, then that thread should be detached.

Think of a detached thread as being the usual sort of thread and reserve nondetached threads for only those situations that require them.

A Simple Threads Example

In Code Example 2-1, one thread executes the procedure at the top, creating a helper thread that executes the procedure `fetch`, which involves a complicated database lookup and might take a while. The mainline thread wants the results of the lookup but has other work to do in the meantime. So it does those other things and then waits for its helper to complete its job by executing `thr_join()`.

The result is passed as a stack parameter, which can be done here because the main thread waits for the spun-off thread to terminate. In general, though, it is better to `malloc`(3C) storage from the heap instead of passing an address to thread stack storage.

Code Example 2-1 A Simple Threads Program

```
void mainline (...) {
       char int result;
       thread_t helper;
       int status;

       thr_create(0,0, fetch, &result,0, &helper);

           /* do something else for a while */

       thr_join(helper, 0, &status);
       /* it's now safe to use result */
   }

   void fetch(int *result) {

           /* fetch value from a database */

       *result = value;
       thr_exit(0);
   }
```

Maintain Thread-Specific Data

Single-threaded C programs have two basic classes of data—local data and global data. For multithreaded C programs a third class is added—thread-specific data (TSD). This is very much like global data, except that it is private to a thread.

Thread-specific data is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a key that is global to all threads in the process. Using the key, a thread can access a pointer (`void` *) that is maintained per-thread.

Maintain thread-specific data with the following three functions.

- `thr_keycreate`()—Create a key specific to the process threads.
- `thr_setspecific`()—Bind a thread value to the key.
- `thr_getspecific`()—Store the value in a specific location.

thr_keycreate(3T)

thr_keycreate() allocates a key that is used to identify thread-specific data in a process. The key is global to all threads in the process, and all threads initially have the value NULL associated with the key when it is created.

Once a key has been created, each thread can bind a value to the key. The values are specific to the binding thread and are maintained for each thread independently.

```
#include <thread.h>

int thr_keycreate(thread_key_t *keyp,
 void (*destructor) (void *value)) ;
```

When thr_keycreate() returns successfully, the allocated key is stored in the location pointed to by *keyp*. The caller must ensure that the storage and access to this key are properly synchronized.

An optional destructor function, *destructor*, can be associated with each key. When 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* when the thread exits. The order in which the *destructor* functions are called is unspecified.

Return Values—thr_keycreate() returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, thr_keycreate() fails and returns the corresponding value.

EAGAIN The key name space is exhausted.

ENOMEM Not enough memory is available.

thr_setspecific(3T)

```
#include <thread.h>

int thr_setspecific(thread_key_t key, void *value);
```

thr_setspecific() binds *value* to the thread-specific data key, *key,* for the calling thread.

Return Values—thr_setspecific() returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, thr_setspecific() fails and returns the corresponding value.

ENOMEM Not enough memory is available.

EINVAL *key* is invalid.

thr_getspecific(3T)

```
#include <thread.h>

int thr_getspecific(thread_key_t key, void **valuep);
```

thr_getspecific() stores the current value bound to *key* for the calling thread into the location pointed to by *valuep*.

Return Values—thr_getspecific() returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, thr_getspecific() fails and returns the corresponding value.

EINVAL *key* is invalid.

Global and Private Thread-Specific Data

Code Example 2-2 shows an excerpt from a multithreaded program. This code is executed by any number of threads, but it has references to two global variables, `errno` and `mywindow`, that really should be references to items private to each thread.

Code Example 2-2 Thread-Specific Data—Global but Private

```
body() {
    ...

    while (write(fd, buffer, size) == -1) {
        if (errno != EINTR) {
            fprintf(mywindow, "%s\n", strerror(errno));
            exit(1);
        }
    }

    ...

}
```

References to `errno` should get the system error code from the system call called by this thread, not by some other thread. So, references to `errno` by one thread refer to a different storage location than references to `errno` by other threads.

The `mywindow` variable is intended to refer to a `stdio` stream connected to a window that is private to the referring thread. So, as with `errno`, references to `mywindow` by one thread should refer to a different storage location (and, ultimately, a different window) than references to `mywindow` by other threads. The only difference here is that the threads library takes care of `errno`, but the programmer must somehow make this work for `mywindow`.

The next example shows how the references to `mywindow` work. The preprocessor converts references to `mywindow` into invocations of the `_mywindow` procedure.

This routine in turn invokes `thr_getspecific()`, passing it the `mywindow_key` global variable (it really is a global variable) and an output parameter, `win`, which receives the identity of this thread's window.

Code Example 2-3 Turning Global References Into Private References

```
#define mywindow _mywindow()

thread_key_t mywindow_key;

FILE *_mywindow(void) {
    FILE *win;

    thr_getspecific(mywindow_key, &win);
    return(win);
}


void thread_start(...) {
    ...
    make_mywindow();
    ...
}
```

The `mywindow_key` variable identifies a class of variables for which each thread has its own private copy; that is, these variables are thread-specific data. Each thread calls `make_mywindow()` to initialize its window and to arrange for its instance of `mywindow` to refer to it.

Once this routine is called, the thread can safely refer to `mywindow` and, after `_mywindow`, the thread gets the reference to its private window. So, references to `mywindow` behave as if they were direct references to data private to the thread.

Code Example 2-4 shows how to set this up.

Code Example 2-4 Initializing the Thread-Specific Data

```
void make_mywindow(void) {
    FILE **win;
    static int once = 0;
    static mutex_t lock;

    mutex_lock(&lock);
    if (!once) {
        once = 1;
        thr_keycreate(&mywindow_key, free_key);
    }
    mutex_unlock(&lock);

    win = malloc(sizeof(*win));
    create_window(win, ...);

    thr_setspecific(mywindow_key, win);
}

void free_key(void *win) {
    free(win);
}
```

First, get a unique value for the key, `mywindow_key`. This key is used to identify the thread-specific class of data. So, the first thread to call `make_mywindow` calls `thr_keycreate`(), which assigns to its first argument a unique key. The second argument is a destructor function that is used to deallocate a thread's instance of this thread-specific data item once the thread terminates.

The next step is to allocate the storage for the caller's instance of this thread-specific data item. Having allocated the storage, a call is made to the `create_window` routine, which sets up a window for the thread and sets the storage pointed to by `win` to refer to it. Finally, a call is made to `thr_setspecific`(), which associates the value contained in `win` (that is, the location of the storage containing the reference to the window) with the key.

After this, whenever this thread calls `thr_getspecific`(), passing the global key, it gets the value that was associated with this key by this thread when it called `thr_setspecific`().

When a thread terminates, calls are made to the destructor functions that were set up in `thr_keycreate`(). Each destructor function is called only if the terminating thread established a value for the key by calling `thr_setspecific`().

Create a Thread—Advanced Features

`thr_create`*(3T)*

```
#include <thread.h>

int thr_create(void *stack_base, size_t stack_size,
    void *(*start_routine) (void *), void *arg, long flags,
    thread_t *new_thread);

size_t thr_min_stack(void);
```

stack_base—Contains the address for the stack that the new thread uses. When *stack_base* is `NULL` then `thr_create`() allocates a stack for the new thread with at least *stack_size* bytes.

stack_size—Contains the size, in number of bytes, for the stack that the new thread uses. If *stack_size* is zero, a default size is used. If *stack_size* is not zero, it must be greater than the value returned by `thr_min_stack`().

start_routine—Contains the function with which the new thread begins execution. When *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine* (see `thr_exit`(3T)).

A stack of minimum size might not accommodate the stack frame for *start_routine*, so if a stack size is specified it must provide for the minimum requirement plus room for the *start_routine* requirements and for the functions that *start_routine* calls.

arg—argument can be anything that is described by `void`, which is typically any 4-byte value. Anything larger must be passed indirectly by having the argument point to it.

Note that you can supply only one argument. To get your procedure to take multiple arguments, encode them as one (such as by putting them in a structure).

flags—Specifies attributes for the created thread. In most cases you want to supply a zero to the *flags* argument.

The value in *flags* is constructed from the bitwise inclusive OR of the following:

`THR_SUSPENDED`—Suspends the new thread and does not execute *start_routine* until the thread is started by `thr_continue`(). Use this to operate on the thread (such as changing its priority) before you run it. The termination of a detached thread is ignored.

`THR_DETACHED`—Detaches the new thread so that its thread ID and other resources can by reused as soon as the thread terminates. Set this when you do not want to wait for the thread to terminate.

Note – When there is no explicit synchronization to prevent it, an unsuspended, detached thread can die and have its thread ID reassigned to another new thread before its creator returns from `thr_create`().

`THR_BOUND`—Permanently binds the new thread to an LWP (the new thread is a *bound thread*).

`THR_NEW_LWP`—Increases the concurrency level for unbound threads by one. The effect is similar to incrementing concurrency by one with `thr_setconcurrency`(3T), although `THR_NEW_LWP` does not affect the level set through the `thr_setconcurrency`() function. Typically, `THR_NEW_LWP` adds a new LWP to the pool of LWPs running unbound threads.

When you specify both `THR_BOUND` and `THR_NEW_LWP`, two LWPs are typically created—one for the bound thread and another for the pool of LWPs running unbound threads.

`THR_DAEMON`—Marks the new thread as a daemon. The process exits when all nondaemon threads exit. Daemon threads do not affect the process exit status and are ignored when counting the number of thread exits.

A process can exit either by calling `exit`(2) or by having every thread in the process that was not created with the `THR_DAEMON` flag call `thr_exit`(3T). An application, or a library it calls, can create one or more threads that

should be ignored (not counted) in the decision of whether to exit. The `THR_DAEMON` flag identifies threads that are not counted in the process exit criterion.

new_thread—Points to a location (when *new_thread* is not NULL) where the ID of the new thread is stored when `thr_create`() is successful. The caller is responsible for supplying the storage this argument points to. The ID is valid only within the calling process.

If you are not interested in this identifier, supply a zero value to *new_thread*.

About Stacks

Typically, thread stacks allocated by `thr_create`() begin on page boundaries and any specified 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 result in sending a `SIGSEGV` signal to the offending thread. Thread stacks allocated by the caller are used as is.

When the caller passes in a pre-allocated stack, that stack cannot be freed until the `thr_join`() call for that thread has returned, even when the thread is known to have exited. Then the process exits with a status equal to the return value.

Generally, you do not need to allocate stack space for threads. The threads library allocates one megabyte of virtual memory for each thread's stack with no swap space reserved. (The library uses the `MAP_NORESERVE` option of `mmap`(2) to make the allocations.)

Each thread stack created by the threads library has a red zone. The library creates the red zone by appending a page to the top of a stack to catch stack overflows. This page is invalid and causes a memory fault if it is accessed. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

Specify stacks or their sizes to `thr_create`() only when you're absolutely certain you know that they are correct. There are very few occasions when it is sensible to specify a stack, its size, or both to `thr_create`(). It is difficult even for an expert to know if the right size was specified. This is because even an ABI-compliant program can't determine its stack size statically. Its size is dependent on the needs of the particular runtime environment in which it executes.

Building Your Own Stack

When you specify the size of a thread stack, be sure to account for the allocations needed by the invoked function and by each function called. The accounting should include calling sequence needs, local variables, and information structures.

Occasionally you want a stack that is a bit different from the default stack. An obvious situation is when the thread needs more than one megabyte of stack space. A less obvious situation is when the default stack is too large. You might be creating thousands of threads and just not have the virtual memory necessary to handle the several gigabytes of stack space that this many default stacks require.

The limits on the maximum size of a stack are often obvious, but what about the limits on its minimum size? There must be enough stack space to handle all of the stack frames that are pushed onto the stack, along with their local variables and so on.

You can get the absolute minimum on stack size by calling `thr_min_stack`(), which returns the amount of stack space required for a thread that executes a null procedure. Useful threads need more than this, so be very careful when reducing the stack size.

You can specify a custom stack in two ways. The first is to supply a `NULL` for the stack location, thereby asking the runtime library to allocate the space for the stack, but to supply the desired size in the stacksize parameter to `thr_create`().

The other approach is to take overall aspects of stack management and supply a pointer to the stack to `thr_create`(). This means that you are responsible not only for stack allocation but also for stack deallocation—when the thread terminates, you must arrange for the disposal of its stack.

When you allocate your own stack, be sure to append a red zone to its end by calling `mprotect`(2).

Return Values—`thr_create`() returns a zero and exits when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, `thr_create`() fails and returns the corresponding value.

EAGAIN A system limit is exceeded, such as when too many LWPs have been created.

ENOMEM Not enough memory was available to create the new thread.

EINVAL *stack_base* is not NULL and *stack_size* is less than the value returned by `thr_min_stack`().

`thr_create`(3T) Example—Code Example 2-5 shows how to create a default thread with a new signal mask (`new_mask`) that is assumed to have a different value than the creator's signal mask (`orig_mask`).

In the example, `new_mask` is set to block all signals except for `SIGINT`. Then 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. When it is masked by the creator, then unmasking the signal opens the creator up to this signal. The other alternative is to have the new thread set its own signal mask in its start routine.

Code Example 2-5 `thr_create`() Creates Thread With New Signal Mask

```
thread_t tid;
sigset_t new_mask, orig_mask;
int error;

(void)sigfillset(&new_mask);
(void)sigdelset(&new_mask, SIGINT);
(void)thr_sigsetmask(SIGSETMASK, &new_mask, &orig_mask):
error = thr_create(NULL, 0, dofunc, NULL, 0, &tid);
(void)thr_sigsetmask(SIGSETMASK, NULL, &orig_mask);
```

Get the Minimal Stack Size

`thr_min_stack`*(3T)*

Use `thr_min_stack`(3T) to get the minimum stack size for a thread.

```
#include <thread.h>

size_t thr_min_stack(void);
```

`thr_min_stack`() returns the amount of space needed to execute a null thread (a null thread is a thread that is created to execute a null procedure).

A thread that does more than execute a null procedure should allocate a stack size greater than the size of `thr_min_stack`().

When a thread is created with a user-supplied stack, the user must reserve enough space to run the thread. In a dynamically linked execution environment, it is difficult to know what the thread minimal stack requirements are.

Most users should not create threads with user-supplied stacks. User-supplied stacks exist only to support applications that require complete control over their execution environments.

Instead, users should let the threads library manage stack allocation. The threads library provides default stacks that should meet the requirements of any created thread.

Get and Set Thread Concurrency Level

`thr_getconcurrency`*(3T)*

Use `thr_getconcurrency`() to get the current value of the desired concurrency level. Note that the actual number of simultaneously active threads can be larger or smaller than this number.

```
#include <thread.h>

int thr_getconcurrency(void)
```

Return Values—`thr_getconcurrency`() always returns the current value for the desired concurrency level.

thr_setconcurrency(3T)

Use thr_setconcurrency() to set the desired concurrency level.

```
#include <thread.h>

int thr_setconcurrency(new_level)
```

Unbound threads in a process might or might not be required to be simultaneously active. To conserve system resources, the threads system ensures by default that enough threads are active for the process to make progress, and that the process will not deadlock through a lack of concurrency.

Because this might 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 can be larger or smaller than *new_level*.

Note that an application with multiple compute-bound threads can fail to schedule all the runable threads if thr_setconcurrency() has not been called to adjust the level of execution resources.

You can also affect the value for the desired concurrency level by setting the THR_NEW_LWP flag in thr_create().

Return Values—thr_setconcurrency() returns a zero when it completes successfully. Any other returned value indicates that an error occurred. When any of the following conditions are detected, thr_setconcurrency() fails and returns the corresponding value.

EAGAIN The specified concurrency level would cause a system resource to be exceeded.

EINVAL The value for *new_level* is negative.

Get and Set Thread Priority

An unbound thread is usually scheduled only with respect to other threads in the process using simple priority levels with no adjustments and no kernel involvement. Its system priority is usually uniform and is inherited from the creating process.

`thr_getprio`*(3T)*

`Use thr_getprio`() to get the current priority for the thread.

```
#include <thread.h>

int thr_getprio(thread_t target_thread, int *pri)
```

Each thread inherits a priority from its creator. `thr_getprio`() stores the current priority, *target_thread*, in the location pointed to by *pri*.

Return Values—`thr_getprio()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When the following condition occurs, `thr_getprio`() fails and returns the corresponding value.

ESRCH *target_thread* cannot be found in the current process.

`thr_setprio`*(3T)*

Use `thr_setprio`() to change the priority of the thread.

```
#include <thread.h>

int thr_setprio(thread_t target_thread, int pri)
```

`thr_setprio`() changes the priority of the thread, specified by *target_thread*, within the current process to the priority specified by *pri*. By default, threads are scheduled based on fixed priorities that range from zero, the least significant, to the largest integer. The *target_thread* will preempt lower priority threads, and will yield to higher priority threads.

Return Values—`thr_setprio()` returns zero after completing successfully. Any other returned value indicates that an error occurred. When any of the following conditions occur, `thr_setprio`() fails and returns the corresponding value.

ESRCH *target_thread* cannot be found in the current process.

EINVAL The value of *pri* makes no sense for the scheduling class associated with the *target_thread*.

Scheduling and the Threads Library

The following `libthread` routines affect thread scheduling.

- `thr_setprio`() and `thr_getprio`()

  These routines alter and retrieve the priority of the *target_thread*, which is the priority used by the scheduler in the user-level threads library, and not the priority used by the operating system to schedule LWPs.

  This priority affects the assignment of threads to LWPs—when there are more runable threads than there are LWPs, the higher-priority threads are given LWPs. The scheduling of threads is preemptive, meaning that when a thread becomes runable and its priority is higher than that of some thread currently assigned to an LWP, and there are no other available LWPs, then the lower-priority thread must relinquish its LWP to the higher-priority thread.

- `thr_suspend`() and `thr_continue`()

  These routines control whether a thread is allowed to run. By calling `thr_suspend`(), you put a thread into the suspended state, meaning that it is set aside and will not be granted an LWP even if one is available. The thread is taken out of this state when some other thread calls `thr_continue`() with the suspended thread as the target. These two routines should be used with care—their effects can be dangerous. For example, the thread being suspended might be holding a lock on a mutex, and suspending it could result in a deadlock.

  A thread can be created in the suspended state by including the `THR_SUSPENDED` flag in the flags parameter of `thr_create`().

- `thr_yield`()

The `thr_yield`() routine causes the calling thread to relinquish its LWP when a thread of equal priority is runable and not suspended. (There cannot be a runable thread of higher priority that is not running, since it would have taken the LWP by preemption.) This routine is of particular importance because there is no time-slicing of threads on LWPs (although, of course, the operating system time-slices the execution of LWPs).

Finally, note that `priocntl`(2) also affects thread scheduling. See “LWPs and Scheduling Classes” on page 84 for more information.

2