

POSIX Threads

Version 1.0

<http://vorlon.ces.cwru.edu/~bjk4/week10/week10.pdf>

Benjamin Karas

April 12, 2000

Contents

1	What are threads?	2
2	Programming	3
2.1	Overview	3
2.1.1	Getting help	4
2.1.2	Compiling	5
2.1.3	Multi-thread unsafe functions	5
2.2	Threads	5
2.2.1	pthread_create(3t)	5
2.2.2	pthread_attr_init(3t)	6
2.3	pthread_attr_destroy(3t)	7
3	Thread management	7
3.1	pthread_self(3t)	7
3.2	pthread_equal(3t)	7
3.3	pthread_join(3t)	7
3.4	pthread_detach(3t)	8
3.5	pthread_cancel(3t)	8
3.6	pthread_exit(3t)	8
4	Mutexes	8
4.1	pthread_mutex_init(3t)	9
4.2	pthread_mutex_destroy(3t)	9
4.3	pthread_mutexattr_init(3t)	9
4.4	pthread_mutexattr_destroy(3t)	9
5	Mutex operations	10
5.1	pthread_mutex_lock(3t)	10
5.2	pthread_mutex_trylock(3t)	10
5.3	pthread_mutex_unlock(3t)	10

6	Condition variables	10
6.1	Overview	10
6.2	pthread_cond_init(3t)	11
6.3	pthread_cond_destroy(3t)	11
6.4	pthread_condattr_init(3t)	11
6.5	pthread_condattr_destroy(3t)	11
7	Condition variable operations	12
7.1	pthread_cond_wait(3t)	12
7.2	pthread_cond_timedwait(3t)	12
7.3	pthread_cond_signal(3t)	13
7.4	pthread_cond_broadcast(3t)	13
8	Read-write locks	13
8.1	pthread_rwlock_init(3t)	13
8.2	pthread_rwlock_destroy(3t)	14
8.3	pthread_rwlockattr_init(3t)	14
8.4	pthread_rwlockattr_destroy(3t)	14
8.5	pthread_rwlock_rdlock(3t)	14
8.6	pthread_rwlock_wrlock(3t)	14
8.7	pthread_rwlock_unlock(3t)	15

1 What are threads?

Threads are flows of execution that can coexist within a single process. When you start a new thread, a new flow of execution is created. This is conceptually similar to forking a new process, yet the details differ greatly.

Each process has at least one thread of execution. If there is more than one thread in a process, then each thread will receive a time-slice of the execution time given to the process by the operating system. The scheduling of threads can be controlled to some extent by the user by choosing different scheduling models, but this is beyond scope of this document.

All threads within a process have access to all dynamic, static, and global variables and memory. They also share file descriptors and user and group ids. Although they are executing the same program, different threads are allowed to execute separate threads may be executing different parts of the code at any given point in time. Each thread owns its own set of registers and program counter.

From W. Richard Stevens', *UNIX Network Programming Vol 2: Interprocess Communications*:

All threads within a process share:

- process instructions,
- most data,

- open files (e.g., descriptors),
- signal handlers and signal dispositions,
- current working directory, and
- user and group IDs.

But each thread has its own

- thread ID,
- set of registers, including program counter and stack pointer,
- stack (for local variables and return addresses),
- `errno`
- signal mask, and
- priority.

Threads are powerful for a two main reasons. First, they are faster than processes because the context switch is less severe. Second, they can share data quickly and easily. However, this ability also causes concurrency problems.

POSIX¹ threads provide several tools to solve concurrency problems:

mutexes These operate just like binary semaphore mutexes. They can be used to protect critical sections of code. Unlike semaphores, they are treated like keys; they are either held by a thread or not. A thread may only lock or unlock a mutex.

condition variables Unrelated to monitors or conditional critical regions, condition variables are used to wake a thread when a condition has changed.

read-write locks Since the readers-writer problem is so common, POSIX threads provide a read-write lock. This lock solves the readers-writer problem.

2 Programming

2.1 Overview

In programming threads, we will deal with the following opaque data types:

pthread_t This is a pthread ID. It uniquely references a single thread within a process.

pthread_attr_t This is a pthread attribute data type. It is used to set options for a thread when you create it. If you do not use this data type, then the system defaults will be used when you create threads.

¹POSIX is the current standard for threads. Other versions exist, but POSIX is the nice one

`pthread_mutex_t` This is a pthread mutex data type. It is used to track information for locking and unlocking critical sections.

`pthread_mutexattr_t` This data type is used to setup options for mutexes.

`pthread_cond_t` This is a pthread condition variable data type. It tracks information concerning condition variables for you.

`pthread_condattr_t` Use this data type to setup options for condition variables.

`pthread_rwlock_t` This is a read write lock data type. It tracks information used to implement read-write locks.

`pthread_rwlockattr_t` Use this data type to setup options for read-write locks.

Each of the above data types is opaque. This means we don't care about what is in them, we just use them in functions designed to operate on the variables.

Each data type must be initialized before it can be operated on. In initializing mutexes, condition variables, or read-write locks, you can optionally use an attribute variable to customize options associated with the variable.

All pthread function calls return zero (0) on success and a positive error value otherwise.

2.1.1 Getting help

The UNIX machines may or may not be setup to access the pthread manual pages by default. First, make sure `/usr/man` or `/usr/share/man` is in your `MANPATH` by typing:

```
echo $MANPATH
```

Next, to access them, try typing something like the following:

```
man -s 3t pthread_create
```

To get a list of topics, type the following:

```
ls /usr/man/sman3t/pthread*
```

Warning: In some cases the manual pages are wrong (e.g. for `pthread_cond_wait`). If you are unsure about the syntax for a function, either look it up in one of the books (The Monkey Book, or *UNIX Network Programming Vol 2: Interprocess Communications* by W. Richard Stevens), or check the function prototype in `/usr/include/pthread.h`.

2.1.2 Compiling

To compile programs with threads, you need to include the `libpthread` library using the `-lpthread` option for `gcc`. If you continue to get errors, make sure you include the necessary headers and `#defines`.

You should `#define _REENTRANT` before you include any header files. Some functions do not work well in multi-threaded environments. Those that are safe say so in the manual page, such as that for `rand_r(3C)`.

2.1.3 Multi-thread unsafe functions

Some system calls do not support multiple threads. This is a short list (from the man page `attributes(5)`):

Unsafe	Safe
<code>ctime</code>	<code>ctime_r</code>
<code>localtime</code>	<code>localtime_r</code>
<code>asctime</code>	<code>asctime_r</code>
<code>gmtime</code>	<code>gmtime_r</code>
<code>ctermid</code>	<code>ctermid_r</code>
<code>getlogin</code>	<code>getlogin_r</code>
<code>rand</code>	<code>rand_r</code>
<code>readdir</code>	<code>readdir_r</code>
<code>strtok</code>	<code>strtok_r</code>
<code>tmpnam</code>	<code>tmpnam_r</code>

2.2 Threads

2.2.1 `pthread_create(3t)`

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

This function spawns a new thread within the process.

The first parameter is used to pass back the thread ID of the new thread to the caller.

The options for the thread are obtained either from the attribute variable referenced in the second parameter, or from the system defaults if the second parameter is `NULL`.

You specify where the new thread will begin executing using the third parameter. The function *must* return a `void *` and take a `void *` argument. See the code example for the syntax regarding how to do this.

The last parameter is the argument that will be passed to the function given in the third parameter. To be valid, the pointer must point to static, global, or dynamic memory. This is required because each thread has its own stack, and thus its own automatic (local) variables.

2.2.2 pthread_attr_init(3t)

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

This function initializes an attribute object. `pthread_attr_init(3t)` will modify the variable you reference in the first parameter. Be sure to call this function before using a `pthread_attr_t` in any other function.

Once initialized, you can modify a number of attributes using the following functions. Look in the manual pages for descriptions about what these functions do.

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

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

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

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);

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

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

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

int pthread_attr_setstacksize
(pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize
```

```
(const pthread_attr_t *attr, size_t *stacksize);
```

All the above functions return zero (0) on success and a positive error value otherwise.

2.3 pthread_attr_destroy(3t)

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

This function releases any memory associated with an attribute and uninitialized it. If you want to reuse an attribute, reinitialize it after calling this function.

3 Thread management

3.1 pthread_self(3t)

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Use this function to return the thread ID of the current thread.

3.2 pthread_equal(3t)

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

If you store thread IDs globally and you have to figure out which thread you are... use this function to compare thread IDs safely.

3.3 pthread_join(3t)

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

This function is conceptually equivalent to `wait(2)`. The caller will wait until the specified non-detached thread returns, is canceled, or calls `pthread_exit(3t)`. If you want, you can also retrieve the return value.

You *must* give a specific thread ID to `pthread_join(3t)` as the first parameter. This is annoying, but unavoidable given the calling semantics.

The function will return the return value of the exiting thread using the second parameter. To be valid, the exiting thread must return a `void *` to a static, global, or dynamic variable.

3.4 pthread_detach(3t)

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

Detaching a thread prevents its return value from being returned to any other threads that call `pthread_join(3t)`. If you do not detach a thread, its resources will not be freed until another thread calls `pthread_join(3t)`. You can avoid this hassle using `pthread_detach(3t)`.

Give `pthread_detach(3t)` the thread ID of the thread you wish to detach. This does not have to be the current thread's ID.

3.5 pthread_cancel(3t)

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

Canceling a thread causes it to stop executing. Specify the thread ID in the first parameter.

3.6 pthread_exit(3t)

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

A thread may safely exit in any one of three ways:

1. using `return`
2. using `pthread_exit(3t)`
3. using `pthread_cancel(3t)`

Use one of the first two methods if you want to exit the current thread. Use the third method to cancel a separate thread.

The parameter to `pthread_exit(3t)` is the return value for the function. It should be a pointer to a static, global, or dynamic variable so that another thread can access the data it points to.

4 Mutexes

Mutexes are used to protect critical sections. They do this by only allowing one thread to lock a mutex at any point in time. If another thread attempts to lock an already locked mutex, it will block until it becomes unlocked. You can use this property to bracket accesses to shared data to prevent concurrency problems.

Mutexes must be initialized before use, either using a mutex attribute variable or using the system defaults. When you will no longer use a mutex, you should destroy it.

4.1 pthread_mutex_init(3t)

```
#include <pthread.h>
int pthread_mutex_init
    (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER
```

Mutexes must be initialized. To do so, you may either use the function, `pthread_mutex_init(3t)`, or you may assign the variable to the initializer, `PTHREAD_MUTEX_INITIALIZER`.

The first argument is the address of your mutex variable of type `pthread_mutex_t`.

The second parameter may be `NULL`, or point to a previously initialized attribute object.

4.2 pthread_mutex_destroy(3t)

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Destroying a mutex frees the resources associated with the mutex. This will also uninitialized the mutex, so if you want to reuse it, remember to reinitialize it after calling this function.

4.3 pthread_mutexattr_init(3t)

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

Before you can change options in a mutex attribute object, you must initialize it. Give the function the address of a variable of type `pthread_mutexattr_t`.

You can change attributes using the following functions:

```
int pthread_mutexattr_getpshared
    (const pthread_mutexattr_t *attr, int *pshared);
int pthread_mutexattr_setpshared
    (pthread_mutexattr_t *attr, int pshared);
```

The above function return zero (0) on success, and a positive error value otherwise.

4.4 pthread_mutexattr_destroy(3t)

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

This function released resources associated with the mutex attribute. If you want to reuse the attribute variable, remember to reinitialize it after calling this function.

5 Mutex operations

5.1 pthread_mutex_lock(3t)

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Locking a resource is similar to waiting on a binary semaphore. Only one thread may hold mutex at any one point in time. If the mutex is not available, then the caller will block until it is. The parameter is the address of the initialized mutex.

We will see later that mutexes and condition variables work together in some very nice ways, but that might be confusing at first, especially toward how things maintain mutexes.

5.2 pthread_mutex_trylock(3t)

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

This function is similar to `pthread_mutex_lock(3t)`, except that it will not block. Instead, if the mutex is not available, it will return `EBUSY` as an error. The parameter is the address of the initialized mutex.

5.3 pthread_mutex_unlock(3t)

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

This is equivalent to signalling a binary semaphore. This releases the mutex, which lets other threads grab it. The scheduling attribute is used to choose the next thread to wake up, if any are blocked. The parameter is the address of the initialized mutex.

6 Condition variables

6.1 Overview

A condition variable lets a process block on it until another process signals the condition variable. It is assumed that if you are waiting for a condition to occur, such as data arriving for processing, you will repeatedly block until it turns true.

Each condition variable is tied to a mutex. This mutex is used to protect the condition variable. The mutex should be locked prior to waiting on the condition variable and unlocked after waking. In reality, the wait function will unlock the mutex prior to blocking and re-lock it upon waking.

The normal use of condition variables is to wait for a variable to reach a certain value. This is why the condition variable is tied to a mutex.

6.2 `pthread_cond_init(3t)`

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

Condition variables must be initialized. To do so, you may either use the function, `pthread_cond_t(3t)`, or you may assign the variable to the initializer, `PTHREAD_COND_INITIALIZER`. Make sure the condition variable is either in dynamic memory, is static, or is global. Automatic variables, which are kept on the stack, are not shared between threads.

The first parameter is the condition variable to initialize.

You can set the options for the condition variable either using the system defaults by using `NULL` for the second parameter, or using a previously initialized condition variable attribute variable.

6.3 `pthread_cond_destroy(3t)`

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

Destroying a condition variable releases the resources associated with the condition variable. If you plan on reusing the condition variable after calling this function, remember to reinitialize it.

6.4 `pthread_condattr_init(3t)`

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

Before you can change options in a condition variable attribute object, you must initialize it. Give the function the address of a condition variable attribute. Once initialized, you can use the following functions to change options:

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

The above functions return zero (0) on success and a positive error value otherwise.

6.5 `pthread_condattr_destroy(3t)`

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

This function releases resources associated with the condition variable attribute variable. If you plan on reusing the attribute after calling this function, remember to reinitialize it.

7 Condition variable operations

7.1 `pthread_cond_wait(3t)`

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

You use this function to block on a condition variable. Before you can wait on the condition variable, however, you must lock the mutex. This mutex is used to protect the condition variable data structures. Do not worry about the associated mutex causing a deadlock, for the function call will unlock it just prior to blocking and will re-lock it when it unblocks. The manual page for this function is especially well written, even though its function definition is incorrect. I highly recommend you read it for details on how `pthread_cond_wait(3t)` works.

According to the manual page:

```
pthread_cond_wait atomically unlocks the mutex (as per pthread_unlock_mutex)
and waits for the condition variable cond to be signaled. The thread
execution is suspended and does not consume any CPU time until
the condition variable is signaled. The mutex must be locked by the
calling thread on entrance to pthread_cond_wait. Before returning
to the calling thread, pthread_cond_wait re-acquires mutex (as per
pthread_lock_mutex)
```

If you are using this function to wait until a variable reaches a certain state, you should place this function in a loop to avoid problems with spurious awakening.

7.2 `pthread_cond_timedwait(3t)`

```
#include <pthread.h>
int pthread_cond_timedwait
    (pthread_cond_t *cond, pthread_mutex_t *mutex,
    const struct timespec *abstime);
```

This function is very similar to `pthread_cond_wait(3t)`, except that you can define a time limit to the waiting. The time limit is contained in a `struct timespec`, which refers to a specific point in time, as opposed to an offset. The structure looks like:

```

struct timespec {
    time_t  tv_sec;      /* seconds */
    long    tv_nsec;    /* nanoseconds */
}

```

`tv_sec` is the number of seconds since January 1, 1970, UTC.

7.3 `pthread_cond_signal(3t)`

```

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

```

This function wakes one thread that waiting on a condition variable.

7.4 `pthread_cond_broadcast(3t)`

```

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

```

This function wakes all threads waiting on a condition variable.

8 Read-write locks

Read-write locks help solve readers-writer type problems. In these problems, no threads may enter a critical section if a single writer is writing. However, any number of readers may enter if no writer is writing, and no writer may enter if any readers are reading.

These locks work very similarly to mutexes, except that the functions specify the type of action intended by the thread.

8.1 `pthread_rwlock_init(3t)`

```

#include <pthread.h>
int pthread_rwlock_init
    (pthread_rwlock_t *rwlock,
     const pthread_rwlockattr_t *attr);
pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;

```

Read-write locks must be initialized before use. To do so, you may either use the function, `pthread_rwlock_init(3t)`, or you may assign the lock to the initializer, `PTHREAD_RWLOCK_INITIALIZER`. Make sure the read-write lock is located in either dynamic, static, or global memory, as automatic variables will not be shared between threads.

The first parameter is the read-write lock to initialize.

If the second parameter is `NULL`, then the system defaults are used to initialize the read-write lock. Otherwise, the read-write lock attribute variable is used to setup the options of the read-write lock.

8.2 pthread_rwlock_destroy(3t)

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

This function releases the resources associated with the read-write lock. If you plan on reusing the read-write lock variable after calling this function, remember to reinitialize it.

8.3 pthread_rwlockattr_init(3t)

```
#include <pthread.h>
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

Before you can change options in a read-write lock attribute variable, you must initialize it. Give the function the address of your read-write lock attribute variable.

Once initialized, you can use the following functions to change options:

```
int pthread_rwlockattr_getpshared
    (const pthread_rwlockattr_t *attr, int *pshared);
int pthread_rwlockattr_setpshared
    (pthread_rwlockattr_t *attr, int pshared);
```

8.4 pthread_rwlockattr_destroy(3t)

```
#include <pthread.h>
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Destroying a read-write lock attribute variable releases resources associated with it. If you plan on reusing the read-write lock attribute variable after calling this function, remember to reinitialize it.

8.5 pthread_rwlock_rdlock(3t)

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

These functions are equivalent to the functions `pthread_cond_lock(3t)` and `pthread_cond_trylock(3t)`, except they follow the rules imposed on readers by the readers-writer problem.

8.6 pthread_rwlock_wrlock(3t)

```
#include <pthread.h>
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

These functions are equivalent to the functions `pthread_cond_lock(3t)` and `pthread_cond_trylock(3t)`, except they follow the rules imposed on writers by the readers-writer problem.

8.7 `pthread_rwlock_unlock(3t)`

```
#include <pthread.h>
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

This function is equivalent to `pthread_cond_unlock(3t)`, except that it will not violate the conditions imposed by the readers-writer problem.

Benjamin Karas
bjk4@po.cwru.edu