Getting Started With POSIX Threads

Tom Wagner
Don Towsley
Department of Computer Science
University of Massachusetts at Amherst

July 19, 1995

1. Introduction: What is a thread and what is it good for?

Threads are often called *lightweight processes* and while this term is somewhat of an over simplification, it is a good starting point. Threads are cousins to UNIX processes though they are not processes themselves. To understand the distinction we must examine the relation between UNIX processes and Mach tasks and threads. In UNIX, a process contains both an executing program and a bundle of resources such as the file descriptor table and address space. In Mach, a task contains only a bundle of resources; threads handle all execution activities. A Mach task may have any number of threads associated with it and all threads must be associated with some task. All threads associated with a given task share the task's resources. Thus a thread is essentially a program counter, a stack, and a set of registers -- all the other data structures belong to the task. A UNIX process in Mach is modeled as a task with a single thread.

Since threads are very small compared with processes, thread creation is relatively cheap in terms of CPU costs. As processes require their own resource bundle, and threads share resources, threads are likewise memory frugal. Mach threads give programmers the ability to write concurrent applications that run on both uniprocessor and multiprocessor machines transparently, taking advantage of the additional processors when they exist. Additionally, threads can increase performance in a uniprocessor environment when the application performs operations that are likely to block or cause delays, such file or socket I/O.

In the following sections we discuss portions of the POSIX threads standard and its specific implementation in the DEC OSF/1 OS, V3.0. POSIX threads are called *pthreads* and are similar to the non-POSIX *cthreads*.

2. Hello World

Now that the formalities are over with, lets jump right in. The `pthread_create` function creates a new thread. It takes four arguments, a thread variable or holder for the thread, a thread attribute, the function for the thread to call when it starts execution, and an argument to the function. For example:

```
pthread_t          a_thread;
pthread_attr_t     a_thread_attribute;
void               thread_function(void *argument);
```

```
char               *some_argument;

pthread_create( &a_thread, a_thread_attribute, (void *)&thread_function,
                   (void *) &some_argument);
```

A thread attribute currently only specifies the minimum stack size to be used. In the future thread attributes may be more interesting, but for now, most applications can get by simply using the default by passing `pthread_attr_default` in the thread attribute parameter position. Unlike processes created by the UNIX `fork` function that begin execution at the same point as their parents, threads begin their execution at the function specified in `pthread_create`. The reason for this is clear; if threads did not start execution elsewhere we would have multiple threads executing the same instructions *with the same resources*. Recall that processes each have their own resource bundle and threads share theirs.

Now that we know how to create threads we are ready for our first application. Lets design a multi-threaded application that prints the beloved "Hello World" message on `stdout`. First we need two thread variables and we need a function for the new threads to call when they start execution. We also need some way to specify that each thread should print a different message. One approach is to partition the words into separate character strings and to give each thread a different string as its "startup" parameter. Take a look at the following code:

```
void print_message_function( void *ptr );

main()
{
      pthread_t thread1, thread2;
      char *message1 = "Hello";
      char *message2 = "World";

      pthread_create( &thread1, pthread_attr_default,
                        (void*)&print_message_function, (void*) message1);
      pthread_create(&thread2, pthread_attr_default,
                        (void*)&print_message_function, (void*) message2);

      exit(0);
}

void print_message_function( void *ptr )
{
      char *message;
      message = (char *) ptr;
      printf("%s ", message);
}
```

Note the function prototype for `print_message_function` and the casts preceding the message arguments in the `pthread_create` call. The program creates the first thread by calling `pthread_create` and passing "Hello" as its startup argument; the second thread is created with "World" as its argument. When the first thread begins execution it starts at the `print_message_function` with its "Hello" argument. It prints "Hello" and comes to the end of the function. A thread terminates when it leaves its initial function therefore the first thread terminates after printing "Hello." When the second thread executes it prints "World" and likewise terminates. While this program appears reasonable, there are two major flaws.

First and foremost, threads execute concurrently. Thus there is no guarantee that the first thread reaches the `printf` function prior to the second thread. Therefore we may see "World Hello" rather than "Hello World."

There is a more subtle point. Note the call to `exit` made by the parent thread[1] in the main block. If the parent thread executes the `exit` call prior to either of the child threads executing `printf`, no output will be generated at all. This is because the `exit` function exits the process (releases the task) thus terminating all threads. Any thread, parent or child, who calls `exit` can terminate all the other threads along with the process. Threads wishing to terminate explicitly must use the `pthread_exit` function.

Thus our little hello world program has two race conditions. The race for the `exit` call and the race to see which child reaches the `printf` call first. Lets fix the race conditions with a little crazy glue and duct tape. Since we want each child thread to finish before the parent thread, lets insert a delay in the parent that will give the children time to reach `printf`. To ensure that the first child thread reaches `printf` before the second, lets insert a delay prior to the `pthread_create` call that creates the second thread. The resulting code is:

```
void print_message_function( void *ptr );

main()
{
      pthread_t thread1, thread2;
      char *message1 = "Hello";
      char *message2 = "World";

      pthread_create( &thread1, pthread_attr_default,
                        (void *) &print_message_function, (void *) message1);
      sleep(10);
      pthread_create(&thread2, pthread_attr_default,
                        (void *) &print_message_function, (void *) message2);

      sleep(10);
      exit(0);
}

void print_message_function( void *ptr )
{
      char *message;
      message = (char *) ptr;
      printf("%s", message);
      pthread_exit(0);
}
```

Does this code meet our objective? Not really. It is never safe to rely on timing delays to perform synchronization. Because threads are so tightly coupled its tempting to approach them with a less rigorous attitude concerning synchronization, but that temptation must be avoided. The race condition here is exactly the same situation we have with a distributed application and a shared resource. The resource is the standard output and the distributed computing elements are the three threads. Thread one must use `printf`/`stdout` prior to thread two and both must do their business before the parent thread calls `exit`.

Beyond our attempt to synchronize using delays, we have made yet another blunder. The `sleep` function, like the `exit` function relates to processes. When a thread calls `sleep` the entire process sleeps, i.e., all threads sleep while the process sleeps. Thus we have exactly the same situation as we had without the calls to `sleep` but the program takes twenty seconds longer to run. The proper function to use when delaying a thread is `pthread_delay_np` (np stands for *not process*). For example, to delay a thread for two seconds:

---

[1] While all threads are equal, we shall often refer to the single thread that begins execution with the application the *parent thread* to distinguish it from later spawned threads, which we will refer to as *child threads* or *children*.

```
struct timespec delay;
delay.tv_sec = 2;
delay.tv_nsec = 0;
pthread_delay_np( &delay );
```

Functions covered in this section: `pthread_create()`, `pthread_exit()`, `pthread_delay_np()`.

3. Thread Synchronization

POSIX provides two thread synchronization primitives, the mutex and the condition variable. Mutexes are simple lock primitives that can be used to control access to a shared resource. Note that with threads, the entire address space is shared so everything can be considered a shared resource. However, in most cases threads work individually with (conceptually) private local variables, those created within the function called by `pthread_create` and successive functions, and combine their efforts through global variables. Access to the commonly written items must be controlled.

Lets create a readers/writers application where a single reader and a single writer communicate using a shared buffer and access is controlled using a mutex.

```
void reader_function(void);
void writer_function(void);

char buffer;
int buffer_has_item = 0;
pthread_mutex_t mutex;
struct timespec delay;

main()
{
      pthread_t reader;

      delay.tv_sec = 2;
      delay.tv_nsec = 0;

      pthread_mutex_init(&mutex, pthread_mutexattr_default);
      pthread_create( &reader, pthread_attr_default, (void*)&reader_function,
                        NULL);
      writer_function();
}

void writer_function(void)
{
      while(1)
      {
            pthread_mutex_lock( &mutex );
            if ( buffer_has_item == 0 )
            {
                  buffer = make_new_item();
                  buffer_has_item = 1;
            }
            pthread_mutex_unlock( &mutex );
            pthread_delay_np( &delay );
      }
}

void reader_function(void)
{
```

```
        while(1)
        {
              pthread_mutex_lock( &mutex );
              if ( buffer_has_item == 1)
              {
                    consume_item( buffer );
                    buffer_has_item = 0;
              }
              pthread_mutex_unlock( &mutex );
              pthread_delay_np( &delay );
        }
  }
```

In this simple program we assume that the buffer can only hold one item so it is always in one of two states, either it has an item or it doesn't. The writer first locks the mutex, blocking until it is unlocked if it is already locked, then checks to see if the buffer is empty. If the buffer is empty, it creates a new item and sets the flag, `buffer_has_item`, so that the reader will know the buffer now has an item. It then unlocks the mutex and delays for two seconds to give the reader a chance to consume the item. This delay is different from our previous delays in that it is only meant to improve program efficiency. Without the delay, the writer will release the lock and in the next statement attempt to regain the lock again with the intent of creating another item. Its very likely that the reader has not had a chance to consume the item so quickly so the delay is a good idea.

The reader takes a similar stance. It obtains the lock, checks to see if an item has been created, and if so consumes the item. It releases the lock and then delays for a short while giving the writer the chance to create a new item. In this example the reader and writer run forever, generating and consuming items. However, if a mutex is no longer needed in a program it should be released using `pthread_mutex_destroy(&mutex)`. Observe that in the mutex initialization function, which is required, we used the `pthread_mutexattr_default` as the mutex attribute. In OSF/1 the mutex attribute serves no purpose what so ever, so use of the default is strongly encouraged.

The proper use of mutexes guarantees the elimination of race conditions. However, the mutex primitive by itself is very weak as it has only two states, locked or unlocked. The POSIX condition variable supplements mutexes by allowing threads to block and await a signal from another thread. When the signal is received, the blocked thread is awaken and attempts to obtain a lock on the related mutex. Thus signals and mutexes can be combined to eliminate the spin-lock problem exhibited by our readers/writers problem. We have designed a library of simple integer semaphores using the pthread mutex and condition variables and will henceforth discus synchronization in that context. The code for the semaphores can be found in Appendix A and detailed information about condition variables can be found in the man pages.

Functions covered in this section: `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_mutex_destroy()`.

4. Coordinating Activities With Semaphores

Let us revisit our readers/writers program using semaphores. We will replace the mutex primitive with the more robust integer semaphore and eliminate the spin-lock problem. Semaphore operations are `semaphore_up`, `semaphore_down`, `semaphore_init`, `semaphore_destroy`, and `semaphore_decrement`. The up and down functions conform to traditional semaphore semantics -- the down operation blocks if the semaphore has a value less than or equal to zero and the up operation increments the semaphore. The init function must be called prior to semaphore use and all semaphores are initialized with

a value of one. The destroy function releases the semaphore if it is no longer used. All functions take a single argument that is a pointer to a semaphore object.

Semaphore decrement is a non-blocking function that decrements the value of the semaphore. It allows threads to decrement the semaphore to some negative value as part of an initialization process. We will look at an example that uses `semaphore_decrement` after the readers/writers program.

```
void reader_function(void);
void writer_function(void);

char buffer;
Semaphore writers_turn;
Semaphore readers_turn;

main()
{
      pthread_t reader;

      semaphore_init( &readers_turn );
      semaphore_init( &writers_turn );

      /* writer must go first */
      semaphore_down( &readers_turn );

      pthread_create( &reader, pthread_attr_default,
                        (void *)&reader_function, NULL);
      writer_function();
}

void writer_function(void)
{
      while(1)
      {
            semaphore_down( &writers_turn );
            buffer = make_new_item();
            semaphore_up( &readers_turn );
      }
}

void reader_function(void)
{
      while(1)
      {
            semaphore_down( &readers_turn );
            consume_item( buffer );
            semaphore_up( &writers_turn );
      }
}
```

This example still does not fully utilize the power of the general integer semaphore. Let us revise the hello world program from Section 2 and fix the race conditions using the integer semaphore.

```
void print_message_function( void *ptr );

Semaphore child_counter;
Semaphore worlds_turn;

main()
{
     pthread_t thread1, thread2;
```

```
        char *message1 = "Hello";
        char *message2 = "World";

        semaphore_init( &child_counter );
        semaphore_init( &worlds_turn );

        semaphore_down( &worlds_turn ); /* world goes second */

        semaphore_decrement( &child_counter ); /* value now 0 */
        semaphore_decrement( &child_counter ); /* value now -1 */
        /*
         * child_counter now must be up-ed 2 times for a thread blocked on it
         * to be released
         *
         */


        pthread_create( &thread1, pthread_attr_default,
                        (void *) &print_message_function, (void *) message1);

        semaphore_down( &worlds_turn );

        pthread_create(&thread2, pthread_attr_default,
                        (void *) &print_message_function, (void *) message2);

        semaphore_down( &child_counter );

        /* not really necessary to destroy since we are exiting anyway */
        semaphore_destroy ( &child_counter );
        semaphore_destroy ( &worlds_turn );
        exit(0);
}

void print_message_function( void *ptr )
{
        char *message;
        message = (char *) ptr;
        printf("%s ", message);
        fflush(stdout);
        semaphore_up( &worlds_turn );
        semaphore_up( &child_counter );
        pthread_exit(0);
}
```

Readers can easily satisfy themselves that there are no race conditions in this version of our hello world program and that the words are printed in the proper order. The semaphore `child_counter` is used to force the parent thread to block until both children have executed the `printf` statement and the following `semaphore_up( &child_counter )`.

Functions covered in this section: `semaphore_init()`, `semaphore_up()`, `semaphore_down()`, `semaphore_destroy()`, and `semaphore_decrement()`.

5. Pragmatics

To compile with pthreads you must include the pthread header file, `#include <pthread.h>` and must link to the pthread library. For example, `cc hello_world.c -o hello_world -lpthreads`

To use the semaphore library you must likewise include its header file and link to the object file or the library.

The DEC pthreads are based on the POSIX IV threads standard, not the POSIX VIII threads standard. The function `pthread_join` allows one thread to wait for another to exit. While this could be used in the hello world program to determine when the children are done instead of our decrement/up semaphore operations, the DEC implementation of `pthread_join` has unreliable behavior if the thread object specified no longer exists. For example, in the code below, if `some_thread` no longer exists, `pthread_join` may cause an error instead of just returning.

```
pthread_t some_thread;
void *exit_status;
pthread_join( some_thread, &exit_status );
```

Other strange errors may occur from functions outside of the thread routines. While these errors are few and far between, some libraries make "uni-process" assumptions. For example, we have experienced intermittent difficulties with the buffered stream I/O functions `fread` and `fwrite` that can only be attributed to race conditions. On the issue of errors, though we did not check the return values of the thread calls in our examples to streamline them, the return values should be consistently checked. Almost all pthread related functions will return -1 on an error. For example:

```
pthread_t some_thread;
if ( pthread_create( &some_thread, ... ) == -1 )
{
      perror("Thread creation error");
      exit(1);
}
```

The semaphore library will print a message and exit on erorrs. Some useful functions not covered in the examples:

`pthread_yield();`	Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
`pthread_t me;` `me = pthread_self();`	Allows a pthread to obtain its own identifier
`pthread_t thread;` `pthread_detach(thread);`	Informs the library that the threads exit status will not be needed by subsequent `pthread_join` calls resulting in better threads performance.

For more information consult the library or the man pages, e.g., `man -k pthread`..

Appendix A - Semaphore Library Code

```
/*
 * File:  semaphore.h
 *
 */
#include <stdio.h>
#include <pthread.h>

#ifndef SEMAPHORES
#define SEMAPHORES

typedef struct Semaphore
{
    int         v;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
}
Semaphore;

int         semaphore_down (Semaphore * s);
int         semaphore_up (Semaphore * s);
void        semaphore_destroy (Semaphore * s);
void        semaphore_init (Semaphore * s);
int         semaphore_value (Semaphore * s);
int         tw_pthread_cond_signal (pthread_cond_t * c);
int         tw_pthread_cond_wait (pthread_cond_t * c, pthread_mutex_t * m);
int         tw_pthread_mutex_unlock (pthread_mutex_t * m);
int         tw_pthread_mutex_lock (pthread_mutex_t * m);
void        do_error (char *msg);

#endif
```

```
/*
 * File:  semaphore.c
 *
 */
#include "semaphore.h"


/*
 * function must be called prior to semaphore use -- handles
 * setup and initialization.  semaphore destroy (below) should
 * be called when the semaphore is no longer needed.
 *
 */
void
semaphore_init (Semaphore * s)
{
    s->v = 1;
    if (pthread_mutex_init (&(s->mutex), pthread_mutexattr_default) == -1)
     do_error ("Error setting up semaphore mutex");

    if (pthread_cond_init (&(s->cond), pthread_condattr_default) == -1)
     do_error ("Error setting up semaphore condition signal");
}


/*
 * function should be called when there is no longer a need for
 * the semaphore.  handles deallocation/release.
 *
 */
void
semaphore_destroy (Semaphore * s)
```

```
{
    if (pthread_mutex_destroy (&(s->mutex)) == -1)
     do_error ("Error destroying semaphore mutex");

    if (pthread_cond_destroy (&(s->cond)) == -1)
     do_error ("Error destroying semaphore condition signal");
}


/*
 * function increments the semaphore and signals any threads that
 * are blocked waiting a change in the semaphore.
 *
 */
int
semaphore_up (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));

    (s->v)++;
    value_after_op = s->v;

    tw_pthread_mutex_unlock (&(s->mutex));
    tw_pthread_cond_signal (&(s->cond));

    return (value_after_op);
}


/*
 * function decrements the semaphore and blocks if the semaphore is
 * <= 0 until another thread signals a change.
 *
 */
int
semaphore_down (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
    while (s->v <= 0)
    {
     tw_pthread_cond_wait (&(s->cond), &(s->mutex));
    }

    (s->v)--;
    value_after_op = s->v;

    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}


/*
 * function does NOT block but simply decrements the semaphore.
 * should not be used instead of down -- only for programs where
 * multiple threads must up on a semaphore before another thread
 * can go down, i.e., allows programmer to set the semaphore to
 * a negative value prior to using it for synchronization.
 *
 */
int
semaphore_decrement (Semaphore * s)
{
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
```

```
    s->v--;
    value_after_op = s->v;
    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}


/*
 * function returns the value of the semaphore at the time the
 * critical section is accessed.  obviously the value is not guarenteed
 * after the function unlocks the critical section.  provided only
 * for casual debugging, a better approach is for the programmar to
 * protect one semaphore with another and then check its value.
 * an alternative is to simply record the value returned by semaphore_up
 * or semaphore_down.
 *
 */
int
semaphore_value (Semaphore * s)
{
    /* not for sync */
    int         value_after_op;

    tw_pthread_mutex_lock (&(s->mutex));
    value_after_op = s->v;
    tw_pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
}



/* ------------------------------------------------------------------------ */
/* The following functions replace standard library functions in that   */
/* they exit on any error returned from the system calls.  Saves us     */
/* from having to check each and every call above.                      */
/* ------------------------------------------------------------------------ */


int
tw_pthread_mutex_unlock (pthread_mutex_t * m)
{
    int         return_value;

    if ((return_value = pthread_mutex_unlock (m)) == -1)
     do_error ("pthread_mutex_unlock");

    return (return_value);
}

int
tw_pthread_mutex_lock (pthread_mutex_t * m)
{
    int         return_value;

    if ((return_value = pthread_mutex_lock (m)) == -1)
     do_error ("pthread_mutex_lock");

    return (return_value);
}

int
tw_pthread_cond_wait (pthread_cond_t * c, pthread_mutex_t * m)
{
    int         return_value;

    if ((return_value = pthread_cond_wait (c, m)) == -1)
     do_error ("pthread_cond_wait");
```

```
    return (return_value);
}

int
tw_pthread_cond_signal (pthread_cond_t * c)
{
    int         return_value;

    if ((return_value = pthread_cond_signal (c)) == -1)
     do_error ("pthread_cond_signal");

    return (return_value);
}


/*
 * function just prints an error message and exits
 *
 */
void
do_error (char *msg)
{
    perror (msg);
    exit (1);
}
```