

Getting Started With POSIX Threads

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

July 19, 1995

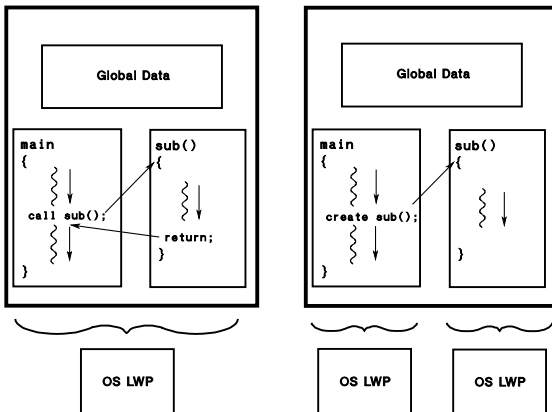
Modified for use in CS341 - Shashikanth Patlolla and Bob Rinker
University of Idaho

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. A process contains a bundle of resources such as the file descriptor table and address space, and one or more threads; the threads handle all execution activities. All threads associated with a process 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 process.

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. 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 a few of the subroutines contained in the POSIX threads standard library, called *pthreads*. It uses a familiar subroutine-like model to implement threads. The thread code itself is placed in a subroutine. The main program, instead of simply calling the subroutine, uses a pthread library call to "activate" the thread. The main program returns from the call immediately, and the thread begins executing on its own, concurrently with the main program. The diagram below illustrates the difference between a subroutine and a thread.



2 Hello World

Now that the formalities are over with, let's jump right in. The `pthread_create` function creates a new thread. It takes four arguments: a thread variable or holder for the thread, a pointer to a thread attribute (which uses default values if set to NULL), the function for the thread to call when it starts execution, and an argument to the function. For example:

```
pthread_t a_thread;  
void *thread_function(void *argument); // Note function and arg type  
char *some_argument;  
  
pthread_create( &a_thread, NULL, thread_function, (void *) &some_argument);
```

NOTE: If `thread_function` isn't declared as above, then third argument to `pthread_create` should be type cast. Example: `(void *) &thread_function`.

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 threads share the process' resource bundle with all other threads in the process.

Now that we know how to create threads we are ready for our first application. Let's 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\n";  
  
    pthread_create( &thread1, NULL, print_message_function, (void *) message1);  
    pthread_create( &thread2, NULL, print_message_function, (void *) message2);  
  
    exit(0);  
} // END main  
  
void * print_message_function( void *ptr )  
{  
    char *message;  
    message = (char *) ptr;  
    printf("%s ", message);  
} // END print_message_function
```

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 passes "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 (albeit rather silly), there are two major flaws.

First, since threads execute concurrently there is no guarantee that the first thread will reach its `printf` function before the second. 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 thread¹ 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 program has two race conditions: the race for the `exit` call and the race to see which child reaches its `printf` first. Let’s fix the race conditions with a little crazy glue and duct tape. Since we want each child thread to finish before the parent thread, let’s insert a delay in the parent, to give the children time to reach `printf`. To ensure that the first child thread reaches `printf` before the second, let’s insert a delay prior to the `pthread_create` call that creates the second thread. The code is:

```
void *print_message_function( void *ptr );

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

    pthread_create( &thread1, NULL, print_message_function, (void *) message1);
    sleep(10);
    pthread_create( &thread2, NULL, print_message_function, (void *) message2);
    sleep(10);
    exit(0);
} // END main

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

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 it’s 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* on most systems. 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 now the program takes twenty seconds longer to run. The proper function to use when delaying a thread is `nanosleep`, which has been specifically implemented to work with threads. For example, to delay a thread for two seconds:

¹While 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;
nanosleep( &delay );
```

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

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 private local variables, i.e., those created within the function called by `pthread_create` and successive functions, and combine their efforts through global (shared) variables. Access to the shared 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 *ptr);
void *writer_function(void *ptr);
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, reader_function, NULL);

    writer_function();
} // END main

void *writer_function(void *ptr)
{
    while(1)
    {
        pthread_mutex_lock( &mutex );
        if ( buffer_has_item == 0 )
        {
            buffer = make_new_item();
            buffer_has_item = 1;
        }
        pthread_mutex_unlock( &mutex );
        nanosleep( &delay );
    }
} // END writer_function
```

```

void *reader_function(void *ptr)
{
    while(1)
    {
        pthread_mutex_lock( &mutex );
        if ( buffer_has_item == 1)
        {
            consume_item( buffer );
            buffer_has_item = 0;
        }
        pthread_mutex_unlock( &mutex );
        nanosleep( &delay );
    }
} // END reader_function

```

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 many implementations the mutex attribute serves no purpose what so ever, so use of the default is strongly encouraged.

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

4 Coordinating Activities With Semaphores

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 awakened 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 semaphore library of simple integer semaphores using the pthread mutex and condition variables and will henceforth discuss 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.

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. The semaphore library includes: `sem_init`, `sem_valinit`, `sem_P`, `sem_V`, and `sem_destroy`, and `sem_value`. The P and V functions conform to traditional semaphore semantics – the P operation blocks if the semaphore has a value less than or equal to zero and the V operation increments the semaphore. One of the init functions must be called prior to semaphore use – `sem_init` initializes the semaphore with a value of one, while `sem_valinit` initializes the semaphore with the value specified by the second argument. The destroy function releases the semaphore if

it is no longer used. Finally, `sem_value` allows you to get the value of the semaphore without blocking or incrementing. In all functions the first argument is a pointer to a semaphore object.

Using the semaphore library, here is a correctly synchronized solution to the reader/writers problem:

```

void *reader_function(void *ptr);
void *writer_function(void *ptr);
void consume_item(char);
char make_new_item(void);

Semaphore writers_turn;
Semaphore readers_turn;

char buffer;

main()
{
    pthread_t reader;

    sem_valinit(&readers_turn, 0 ); /* writer must go first/
    sem_init(&writers_turn );

    pthread_create(&reader, NULL, reader_function, NULL);
    writer_function(NULL);
} // END main

void *writer_function(void *ptr)
{
    while(1)
    {
        sem_P(&writers_turn);
        buffer = make_new_item();
        sem_V(&readers_turn);
    }
} // END writer_function

void *reader_function(void *ptr)
{
    while(1)
    {
        sem_P(&readers_turn);
        consume_item(buffer);
        sem_V(&writers_turn);
    }
} // END reader_function

```

The previous example still does not fully utilize the power of the general integer semaphore. Let's 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";

    sem_valinit( &child_counter, -1);
    sem_init( &worlds_turn);

    sem_P( &worlds_turn ); /* world goes second */
    /*
     * child_counter now must be signalled 2 times for a thread blocked on it
     * to be released
     */
    pthread_create( &thread1, NULL, print_message_function, (void *) message1);

    sem_P( &worlds_turn );

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

    sem_P( &child_counter );

    /* not really necessary to destroy since we are exiting anyway */
    sem_destroy ( &child_counter );
    sem_destroy ( &worlds_turn );

    printf("\n\n");
    exit(0);
} // END main

void * print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s ", message);
    sem_V( &worlds_turn );
    sem_V( &child_counter );
    pthread_exit(0);
} // END print_message_function

```

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 subsequent `sem_V(&child_counter)`.

Functions covered in this section: `sem_init()`, `sem_P()`, `sem_V()`, `sem_destroy()`, and `sem_value()`.

5 Pragmatics

To compile with `pthread`s you must include the `pthread` header file, `#include <pthread.h>` and must link to the `pthread` library. For example,

```
gcc hello_world.c -o hello_world -lpthread
```

To use the semaphore library you must likewise include its header file (i.e., `#include "semaphore.h"`) and link to the object file (or library) when creating the executable:

```
gcc new_hello_world.c semaphore.c -o hello_world -lpthread
```

Each implementation of the `pthread`s library is different, so a threaded program may behave differently when compiled and run on different systems. This is especially true when thread library routines are being used in an "illegal" or unspecified manner. It is always a good idea to write correct code, even when "incorrect" code seems to work.

Other strange errors may occur when routines from outside the `pthread`s library are used within a threaded program. This is because some libraries make "uni-process" assumptions when they were implemented – 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. Libraries that have been written to work properly with threads are called *thread-safe*. It is probably best to assume that a subroutine is not thread-safe unless it is specifically stated as such.

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 errors.

Some useful functions not covered in the examples:

<code>pthread_yield()</code> ;	Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
<code>pthread_t me;</code> <code>me = pthread_self();</code>	Allows a <code>pthread</code> to obtain its own identifier
<code>pthread_t thread;</code> <code>pthread_detach(thread);</code>	Informs the library that the thread's exit status will not be needed by subsequent <code>pthread_join</code> 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 sem_P (Semaphore * s);
int sem_V (Semaphore * s);
void sem_destroy (Semaphore * s);
void sem_init (Semaphore * s);
void sem_valinit (Semaphore * s, int x);
int sem_value (Semaphore * s);
void do_error (char *msg);

#endif

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

/*
 * function must be called prior to semaphore use.
 */
void sem_init (Semaphore * s)
{
    s->v = 1;
    if (pthread_mutex_init (&(s->mutex), NULL) == -1)
        do_error ("Error setting up semaphore mutex");

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

void sem_valinit (Semaphore * s, int x)
{
    s->v = x;
    if (pthread_mutex_init (&(s->mutex), NULL) == -1)
        do_error ("Error setting up semaphore mutex");

    if (pthread_cond_init (&(s->cond), NULL) == -1)
        do_error ("Error setting up semaphore condition signal");
} // END sem_valinit
```

```
/*
 * function should be called when there is no longer a need for
 * the semaphore.
 */
void sem_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");
} // END sem_destroy

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

    pthread_mutex_lock (&(s->mutex));

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

    pthread_mutex_unlock (&(s->mutex));
    pthread_cond_signal (&(s->cond));

    return (value_after_op);
} // END sem_V

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

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

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

    pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
} // END sem_P

/*
 * function returns the value of the semaphore at the time the
 * critical section is accessed. obviously the value is not guaranteed
```

```
* after the function unlocks the critical section. provided only
* for casual debugging, a better approach is for the programmer to
* protect one semaphore with another and then check its value.
* an alternative is to simply record the value returned by semV
* or semP.
*/
int sem_value (Semaphore * s)
{
    /* not for sync */
    int value_after_op;

    pthread_mutex_lock (&(s->mutex));
    value_after_op = s->v;
    pthread_mutex_unlock (&(s->mutex));

    return (value_after_op);
} // END sem_value

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