A Short Introduction to POSIX Threads

Keith Gaughan

March 22, 2003

Contents

1 A bit of background

Many moons ago, when dinosaurs roamed the earth and we were all still living in caves (around 1990, as it happens), most computers were only capable of running one program at a time. This, as you might guess, became rather painful at times. It wasn't too nice having to quit *WordPerfect 5.1* just so that you could check some numbers in *Lotus 1-2-3*. Systems like these were known as *Single Tasking*, or *Single Processing*.

It wasn't the same everywhere though. Back when fish decided it might be fun to try that whole whacky walking-on-land thing (the late '60s and early '70s), there were already Operating Systems capable of running more than one program at once. These were known as *Timesharing Operating Systems*, because they ran on large Mainframes that served a whole bunch of people who were running programs on them. These systems were the precursors to the early *Multitasking* OSs, such as UNIX.

With OSs such as UNIX, you were now able to run multiple programs on the one machine simultaneously, each executing instance being known as a *process*. Now you could spend all day playing Patience whilst the webserver you were running on your machine was serving pictures of your beloved Bonzo the Wonderdog to an ever-so-interested world.

Even with multiprocessing, you're still left with a problem: processes can only do one thing at a time. Say you're browsing the web and you decide to go to Happy Tree Friends[1] to look at all the happy woodland animals being mercilessly slaughtered. If your browser was only capable of doing one thing at a time, you'd have to wait until the whole page was downloaded, then wait for the browser to work out how to render the page, and then, and only then, could watch ants inflict some serious GBH on an anteater.

One way of getting past this difficulty is *multithreading*. Normal processes have only one thread of control, and so can only do one thing at a time. With multithreading, a process can do more than one thing simultaneously, for instance one thread could be taking care of the GUI whilst another is doing some I/O, and yet another is doing some rather heavy calculations.

1.1 What's a thread?

I'm presuming you already know what a process is. Well, a thread is kind of a lightweight process, but unlike a regular heavyweight process, a thread has no memory or resources of its own besides a stack, and its own set of registers. Each heavyweight process consists of at least one thread, and all the other resources it needs such as memory, file descriptors, and so on. Any threads within a process share all these resources with one another. This makes creating and destroying threads a lot less time-consuming than processes.

There are some really good reasons for writing multithreaded programs:

- Increased application responsiveness, e.g. one thread can be handling the application's UI whilst others are doing all the legwork in the background.
- Increased application throughput.
- More efficient use of system resources such as memory and CPU time.
- The ability to run well on uniprocessor and multiprocessor systems.
- Being able to structure your program in such a way that things that can run in parallel can be properly modularised, making for better strutured programs.

And there's far more besides that.

1.2 Is there a downside to using threads?

There's one really nasty side-effect one can run into when writing multithreaded applications—quite horrid things can happen when two threads attempt to access the same shared resources. In fact, horrid is an

[1] Me? Evil? Naw!

understatement, grotesque might be a more apt choice, and after you've tried debugging a misbehaving multithreaded application, you'll know exactly what I mean.

Then there's the horror that are *race conditions*, but we'll deal with them later.

On Linux at least, there's little benefit to using threads over child processes. Unlike operating systems like Solaris where it takes about thirty times longer to spawn a new process than to spawn a new thread, Linux processes spawn almost as quickly as threads, and that's fast. In fact, the only reason why the Apache webserver was rewritten to support threads rather than a process pool was to make it stable under Windows.

Finally, it's worth bearing in mind that multithreaded applications are a royal pain to debug. If you want to know why, read a book on Chaos Theory[2]. About 99% of applications aren't worth multithreading either because they don't lend themselves to it without a modicum of difficulty, or because they just won't benefit from any extra concurrency. Think carefully before you decide to make a program multithreaded[3].

2 POSIX threads

The standard API[4] used for implementing multithreaded applications is *POSIX Threads*, and that's the library this article deals with. As far as threading libraries go, Pthreads is quite a simple one. There's no wierd and wonderful features such as *spin locks* or *priority-inheriting mutexes*. There's not many calls in there either. In fact, there's only about ten or so that you'd use on a regular basis.

To use Pthreads, you'll have to include the following:

```
#include <pthread.h>
```

...and when you're linking, don't forget `-lpthread` so it's linked to the library, e.g.

```
gcc foo.c -o foo -Wall -lpthread
```

2.1 Working with threads

Each process has at least one thread, that created when the `main()` function is called. This thread is free to create additional threads as the programmer sees fit. Here's a simple example.

```
/* Simple Pthreads example. */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void* ThreadFunc(void* arg);

int main(int argc, char argv[])
{
    pthread_t idThread;

    puts("Let's create a thread!");
    pthread_create(&idThread, NULL, ThreadFunc, (void*) 5);
    pthread_join(idThread, NULL);
}

void* ThreadFunc(void* arg)
```

[2] In a nutshell, the interactions of simple entities within a simple system can lead to complex behaviour, or misbehaviour as the case may be.

[3] It's worth bearing in mind that GUI toolkits are something that are regularly multithreaded, AWT being an example for those familiar with Java. This does not mean that the programs using such libraries should be considered multithreaded.

[4] That's *Application Programming Interface*, now wake up you at the back!

```
{
    int i, n;

    /* Get the value of the argument passed in. */
    n = (int) arg;

    /* Do stuff! */
    for (i = 0; i < n; i++)
        printf("Loop %d: La la la!\n", i + 1);

    return NULL;
}
```

This will produce the following output:

```
Let's create a thread!
Loop 1: La la la!
Loop 2: La la la!
Loop 3: La la la!
Loop 4: La la la!
Loop 5: La la la!
```

2.2 Breakdown of the example

This example covers quite a bit. Here's a synopsis of the functions and types used.

2.2.1 pthread_t

Pthreads library includes the type `pthread_t` for holding a thread's id. Any threads that want to manipulate another need to know its id before they can fiddle with it. Needless to say, it could contain anything, so don't go poking at its contents.

```
pthread_t idThread;
```

2.2.2 pthread_create()

This spawns a new thread.

```
int pthread_join(pthread_t* id, const pthread_attr_t* attr,
                 void* (*ThreadFunc)(void*), void* arg);
```

id is used for returning the id of the created thread. With this, the spawning thread (usually the main thread) can manipulate its settings.

attr is a pointer to a `pthread_attr_t` structure. What this does and contains is rather esoteric, so passing `NULL` will cause the default values to be used, which are virtually always what you want anyway.

ThreadFunc is the function the thread should execute. This function's prototype should look like the function `ThreadFunc` in the example.

arg is the value ot pass as an argument to the thread function. In this example, we want to pass the value *5* into it, so we have to cast it into a pointer to void. When the thread function is called, it can cast this value back into whatever it should be. If you want to pass more than one argument to a thread, you'll have to create a structure to hold them.

2.2.3 pthread_join()

Here, we're *joining* a thread. Joining ensures that when the program exits, it waits for the thread to finish executing. This is quite similar to the `wait()` call used for child processes. In practice, you should rarely need to do this.

```
int pthread_join(pthread_t id, void** status);
```

id is the id of the thread we wish to join `pthread_create()`.

status will hold the value returned in `pthread_exit()`, when it's called. If this is `NULL`, it's ignored.

2.3 Other calls you need to know about

2.3.1 pthread_exit()

Terminates the calling thread, returning `status`. If there's nothing you want to pass back, just pass in `NULL`.

The example didn't need this because the `return NULL` at the end does much the same thing.

```
void pthread_exit(void* status);
```

2.3.2 pthread_self()

Returns the id of the current thread.

```
pthread_t pthread_self(void);
```

2.3.3 pthread_yield()

Causes the thread to yield execution in favour of another thread with the same priority.

```
void pthread_yield(void);
```

3 Synchronisation

Think back to when you were younger. They were more innocent days; days when all you had to worry about was counting how many cornflakes had ended up in your Operating Systems lecturer's beard. You concentrate harder and vague recollections about odd things such as *critical sections*, *mutexes*, *monitors*, and *semaphores* bubble up to the surface. If you didn't think knowing about them would be of much use to you, be prepared for a terrible shock.[5]

3.1 Critical sections

When you're dealing with multithreaded code, your threads share virtually everything. It's like living in a house where you're the only person who buys milk and everybody else drinks it on you[6]. Critical sections are like when you're heading to the fridge to get the milk and you want to make sure nobody else goes and nicks it on you in the meantime.

Critical sections are parts of your code where a thread accesses a shared resource can occur. If two or more threads attempt to access the same resource or set of resources, things can get a bit like a bit like a crowd trying to get through the same door at the same time. They must therefore be treated *atomically*—any other threads trying to execute a critical section will be blocked until the lock on that critical section is released.

Critical sections should be kept as short as possible, and carefully optimised because they have a significant effect on our program's performance.

[5] I think the pertainent question here would be "which one of *The Young Ones* do you associate yourself with most?"

[6] I should know, I do! More fool me.

Thread 1	Thread 2
`bal = GetBalance(acc);`	`bal = GetBalance(acc);`
`bal += bal * rate;`	`bal += deposit;`
`/* La la la... */`	`SetBalance(acc, bal);`
`SetBalance(acc, bal);`	`/* D'oh! */`

Figure 1: Why you need synchronisation

3.2 Mutexes

Threads are like being given an Uzi—part of the trick is not to shoot yourself in the foot, or anywhere else for that matter, while you have it.

Mutual exclusion locks, or *mutexes*, are the simplest and most primitive way of delimiting critical sections so that threads behave nicely to one another and Pthreads supplies a family of calls for using them.

The two most important calls are `pthread_mutex_lock()`, which locks a mutex, and the crypically titled `pthread_mutex_unlock()`, which I won't bother explaining[7]. The first thread to lock the mutex in question gets ownership and all other threads are forced to got to sleep. When the owner unlocks it, one of the sleeping threads will be reactivated and given a chance to get ownership. Of course, this could all go rather *Nelson Muntz*[8] and another thread could have managed to get it first. Though normally such behaviour is fine, this might be a problem if you're dealing with realtime systems...[9]

Always acquire mutexes in the same order. It's also a good idea to release them in the reverse of the order you acquired them in.

I think an example's in order!

```
/* Pthreads mutex example. */

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

void* ThreadFunc(void* arg);

/* Create and initialise the mutex for use. */
pthread_mutex_t cntrMutex = PTHREAD_MUTEX_INITIALIZER;

/* The global resource our mutex is to protect. */
int cntr;

int main(int argc, char argv[])
{
    pthread_t idThread1;
    pthread_t idThread2;

    puts("Let's create some threads!");
    pthread_create(&idThread1, NULL, ThreadFunc, (void*) 21);
    pthread_create(&idThread2, NULL, ThreadFunc, (void*) 14);
    pthread_join(idThread1, NULL);
    pthread_join(idThread2, NULL);
}
```

[7] Ok, ok! It unlocks the mutex. Happy now?

[8] Haw haw!

[9] ...which happens to be what I'm going to be doing for my work placement.

```
void* ThreadFunc(void* arg)
{
    int i, nMax, n;

    /* Get the value of the argument passed in. */
    nMax = (int) arg;

    /* Do stuff! */
    for (i = 0; i < nMax; i++)
    {
        n = rand() % nMax;
        pthread_mutex_lock(&cntrMutex);
        for (cntr = 0; cntr < n; i++)
            printf("Loop %d: La la la!\n", cntr + 1);
        pthread_mutex_unlock(&cntrMutex);
    }

    return NULL;
}
```

3.3 Breakdown of the example

In this example, we spawn two threads, each of which competes over a single resource: a global integer called `cntr`. Each recieves a single integer representing the maximum number of times it can execute its loops, `21` for the first thread and `14` for the second. They both compete for access to this so that they can run an inner loop that requires `cntr`.

3.3.1 pthread_mutex_t

This represents a single mutex used for protecting a single resource. Before this can be used, it must be initialised with `pthread_mutex_init()`.

In the example, resource being protected is the global variable `cntr`, but could be anything the threads share, e.g. a file descriptor, access to the screen, etc…

```
pthread_mutex_t cntrMutex;
```

3.3.2 PTHREAD_MUTEX_INITIALIZER

Assigning this macro to a mutex initialises it for use.

```
pthread_mutex_t cntrMutex = PTHREAD_MUTEX_INITIALIZER;
```

3.3.3 pthread_mutex_lock()

Locks the mutex. If it's already locked, the calling thread is suspended until the mutex is unlocked.

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

3.3.4 pthread_mutex_unlock()

Unlocks the mutex and wakes the first thread sleeping on it.

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

3.4 Other calls you need to know about

3.4.1 pthread_mutex_init()

Initialises a mutex for use. You must do this before you can make any other calls on the mutex. This is only really useful for mutexes allocated on the heap with `malloc()`.

```
int pthread_mutex_init(pthread_mutex_t* mutex,
                       const pthread_mutexattr_t* attr);
```

mutex is a pointer to the mutex to initialise.

attr points to some extra configuration data you need to pass for creating some of the more esoteric mutex variants. Generally, you can just pass `NULL`, which will make it use the defaults, as in the example.

3.4.2 pthread_mutex_destroy()

Destroys the mutex, making it unusable. You only need to call this to clean up mutexes initialised with `pthread_mutex_init()`. This doesn't deallocate the mutex itself, however, and you'll still have to call `free()` for that.

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

3.4.3 pthread_mutex_trylock()

Like `pthread_mutex_lock()`, but if the mutex is already locked, returns immediately with `EBUSY`.

```
int pthread_mutex_trylock(pthread_mutex_t* mutex);
```

3.5 Condition variables

I'll deal with these in an additional appendix.

3.6 Synchronisation problems

3.6.1 Deadlocks

Deadlocks occur where one thread needs another thread to do something before it can proceed, and the second needs the first to do so too. They've gone and got themselves stuck in the proverbial door, neither one able to free themselves. This, you might guess, is a Bad Thing.

The only way to avoid it is to be careful. *Always* acquire locks in the same order. *Always*[10]. There should be very little reason for you not to do otherwise.

3.6.2 Race conditions

Threads have a little side-effect—they introduce an element of nondeterminacy[11] into your otherwise sane and predictible program.

Like deadlocks, they're usually a sign that locks have been taken out of order somewhere in the program.

4 Le Fin

If you've any questions or comments, you'll always be able to get me at <kmgaughan@eircom.net>, though <kgaughan@mcom.cit.ie> should work too. The latest version of this document should be downloadable from either my website or my weblog.

[10] If you don't, expect me to be down at your house ready to whack a soggy haddock against the back of your head.

[11] i.e. randomness—you can't tell what order things will happen in.