Multi-threading

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

The concept of multi-threading is often perceived as being difficult, or complicated, or both where, in reality, it is neither of these, It is only necessary to be familiar with a few simple functions, in order to produce quality multi-threaded code.

Too many programmers try to use every feature in the pthreads manual, with little justification, and end up with an application, which suffers from the un-debuggable nightmare: thread deadlock, which we will discuss in more detail, later.

When a process executes in the normal way, its machine instructions are executed, one after the other, until the kernel decides it has had its fair share of CPU time. At this point, the process puts itself to sleep, and another process is scheduled to run. Since only one instruction sequence, at a time, can be executed by such a process, it is considered to have a single thread of execution through its code or, to be single-threaded.

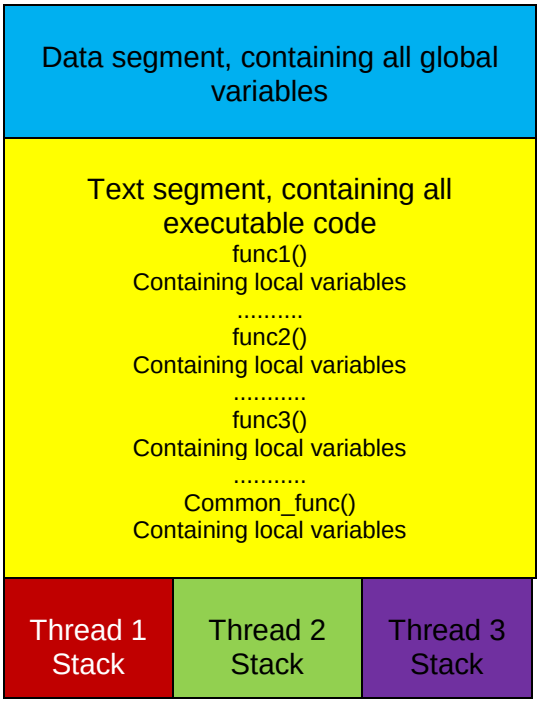
A multi-threaded process, by contrast, simultaneously executes several, perhaps, several hundred instruction sequences, during its scheduled run time and, by virtue of this fact, provides a useful means of improving the throughput of the process.

The Mechanics of Multi-threading

Wherever a thread is created in the source code, the compiler inserts a ‘hook’ into the executable, to tell the kernel that this is a separate path through the code. The granularity of such hooks is at the subroutine level. In other words, a thread may only begin its execution at the start of a subroutine, and not at some arbitrary point in the code.

At run time, when the process executes, the kernel switches the execution path through the process, so as to allow all of the threads to perform their task.

Let us consider a process, which has three parallel threads of execution. The mapping of such a multi-threaded process, in memory, may be represented by the following diagram:



It may be seen that the stack has been split into three independent sections, each of which is allocated to one of the threads of execution. This arrangement, in fact, is the key to the operation of multi-threading, since it provides the means of separating global and local variables.
It may be remembered, that the memory allocation for global variables occurs at compile time, and this memory is consigned to the data segment of the process, shown above.
The memory allocation of variables local to a subroutine, however, occurs at run time, and these variables exist, for the duration of the execution of the subroutine, on the stack. If the stack were common to all threads, then, if a thread entered the subroutine while another thread were executing it, both threads would see the same values of all of the local variables. This is clearly not a desirable state of affairs, since both threads would be free to modify the variables, leaving their final value indeterminate.
The partitioned stack means that this question never arises.

Let us say that we have identified func1(), func2() and func3() as being capable of parallel execution, and wish to create three threads, which will simultaneously execute these functions. Let us also assume, that func1(), func2() and func3() all call common_func(), at some point during their execution.

Before proceeding, let us remind ourselves of what happens during a function call.

Anatomy of a function call

When we make a function call, the following sequence of events occurs:
1. The calling function places its return address on its stack.
2. The calling function places the arguments passed to the called function, one by one, onto its stack, each time incrementing the stack pointer.

3. Execution commences at the address of the called function, which is passed the calling function's stack pointer.
4. The called function strips the arguments, one by one, from the stack, then places any local variables, which it has defined, onto the stack, and initialises them, if necessary.
5. The called function executes, still using the same stack pointer, which it passes to any functions which it may call.
6. When the called function has run to completion, it strips it local variables from the stack (usually by merely incrementing the stack pointer), then places its return value, if any on the stack, and jumps to the return address left there by the calling function.
7. The calling function reads the return value, and continues its execution.

So, all passed parameters and local variables are stored on the stack. This is why, when a thread is created, it is given its own stack, separate from that of any other thread, or the main process, which is now run within a dummy thread, called the 'main thread'. Thus, provided that none of the functions in the calling sequence alters any global variables, each thread can execute in complete oblivion of any other thread.

Thread Safety

Many discussions on threads centre around so-called 'thread safety', and the very expression gives the impression that 'safe' is good, while 'unsafe' is not.

For our purposes, we prefer the term Thread Visibility since, more often than not, the question which faces us, is not a matter of 'Is this variable thread-safe?', but a matter of 'Is this variable visible to all of the threads which need to access it?'
One of the objectives of parallel operation is, frequently, to simultaneously load or manipulate various different members of a common data structure. Such a data structure is, by definition, not thread-safe, and that is exactly what we need.

In summary:
- The value of any variable declared globally, is visible to all threads.
- The value of a local variable of a given function, which has been declared as 'static' is also visible to all threads accessing the function.
- The instantaneous values of all other local variables are only visible to the thread, which is currently executing that function.

.

Thread Creation and Destruction

A thread of execution is created by calling the function pthread_create(), whose syntax is as follows:

Int pthread_create((pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine, void*),void *arg);

This ugly-looking call is complicated by the typedefs used for all of the datatypes.

The 'pthread_t' type is only an int, and the pthread_attr_t type is a data structure describing the thread's attributes, which we will probably never want to change.

In fact, most versions of Unix will only permit processes which run as root to change any attribute other than the stack size, so we can ignore this variable.

The pointer to ‘thread’ is filled by the call with a thread descriptor, unique to that particular thread. Since we never only create one thread (what would be the point?), the ‘thread’ variable is usually an element of an array.

The start_routine is exactly what it says it is, and is the function which the thread will begin to execute, immediately it is created.

The pointer to arg, is an argument of indeterminate type, which we are allowed to pass to our start routine. The fact that there is only one argument permitted, means that, when we are designing such routines, we should make sure that multiple arguments are passed in as pointers to data structures. It is also worth considering that, since the default stack for a thread is of the order of a megabyte, wherever possible, arguments should be passed by reference, to avoid nasty bugs in production.

All of the above is much easier to understand through a concrete example.
First, we need to declare our start_routine, as:

```
void *start_here(void *);
```

and our array of three as-yet-unborn threads as:

```
pthread_t threads[3];
```

If, in reality, our start routine needed three integer arguments, we would declare a structure

```
struct xargs {
   int one;
   int two;
   int three;
};
struct xargs args;
```

Then we would create the first of our threads like this:

```
   if(pthread_create(&threads[0], NULL, start_here, &args) == -1){
      printf(“Thread create failed\n”);
   }
```

When we make this call, two things happen. Firstly, the thread is created and, secondly, it immediately jumps to start_here() and starts to execute it, in much the same way that a call to fork() immediately starts another process.
As with fork(), we need to make an immediate decision, as to whether we should wait for the thread to run to completion, or let it free-run, while we do something else. With threads, however, this decision is complicated by the loss of parallelism resulting from waiting for threads.

Having started our thread, it will run to completion and, at the point in the code where the thread has finished its task, we will need to terminate it gracefully, by calling

```
pthread_exit(void *return_value);
```

which will make available the return value supplied in the return_value pointer to any function waiting for the thread. If the idea of 'void' is a bit alien, think of it as something that can be cast to anything. In reality, it's a char.

If, on the other hand, we would like to kill our thread, from elsewhere in the code, in response, perhaps, to an error condition, we would call

pthread_cancel(pthread_t thread);

Almost all of the pthread calls return 0 on success, and –1 on error, just like Unix system calls. This is a relic of the early days of multi-threading, when threads were actually system calls. Unfortunately, the implications of this, such as operating in kernel mode, with atomic operation, led to too much interference with normal kernel functionality and, these days, threads only partially operate in kernel mode.

Right, so we now have a thread, which dives off to execute its function, as soon as it is created and, presumably, runs to completion, somewhere in the bowels of our code. We will need to know when it has completed its task, otherwise we might exit before it has finished.

Well, in actual fact, the compiler wouldn't just create our one thread: it would create two. The body of our program, where main() executes, would become 'the main thread' which, presumably, would go on to do other things, while the thread we explicitly created ran to completion, somewhere else.

This raises the question of synchronisation.

Thread Synchronisation

As with the creation of processes, the creator has one of two choices: either to wait for the thread to complete, or to let it free-run or, sometimes, both.
At first glance, it may seem as if we are defeating our own objective, by creating a thread, and then waiting for it. In practice, during the execution of a complex program, there may be many changes, from parallel to serial operation, and vice versa.
Consider the case where we are a TCP/IP server, waiting for connections to a socket. It would, obviously, not be practical to wait for the completion of each thread we launched to handle the connections. We would need to let each one run its course, while we waited for the next, otherwise, the server would hang after the first connection.
On the other hand, in the server's graceful-closedown routine, we would expect to have code which waited for all current threads to finish executing, before the server itself quit.

The call to pthread_detach() permits a thread to free-run, while pthread_join() waits for it.

Typical calls would be:

```
if(pthread_detach(threads[0]) == -1){
    printf("Thread failed to detach\n");
}
```

and

```
    if(pthread_join(threads[0], NULL) == -1){
        printf("Thread failed to join\n");
    }
```

The passed-in NULL, is in lieu of a void **return_value pointer, which would have been populated with the return value passed to any pthread_exit() call in the terminating thread.

There only remains one essential pthread function:

```
pthread_t pthread_self()
```

This function returns the identifier of the currently running thread, and is most useful for debugging and logging the progress of the application.
For example,

```
printf("Thread %d executing function fxx\n", pthread_self());
```

Putting it together

We now have all the information we need, to design a multi-threaded application.

Just to demonstrate that threads aren't just smoke and mirrors, (even though they are…) here is a simple test program, which launches 100 threads, each of which sleeps for 10 seconds, or whatever is given as a command line argument, while the main thread waits for all 100. The total run time of this program is, of course, 10 seconds, not 1000.

```
#include <stdio.h>
#include <ctype.h>
#include <pthread.h>
#define __REENTRANT

void *func(void *);

main(argc, argv)                    /* main

int argc;
char **argv;

{

pthread_t thr[100];
int delay;
int i;

if(argc < 2){
    delay = 10;
} else {
    delay = atoi(argv[1]);
}
for(i = 0; i < 100; i++){
    if((pthread_create(&thr[i], NULL, func,  (void *)&delay)) != 0){
        printf("Failed to create thr[%d]\n",i);
```

```
        }
    }
    for(i = 0; i < 100; i++){
        if(pthread_join(thr[i], NULL) != 0){
            printf("Failed to start thr[%d]\n",i);
        }
    }


    printf("All threads terminated\n");

}                       /* main */

void *
func(delay)                     /* func */

void *delay;

{

    printf("Starting thread %d for %d sex..\n", pthread_self(), *(int *)delay);
    sleep(*(int *)delay);
    printf("Thread %d returning..\n", pthread_self());


}                       /* func */
```

The Mutex

Real life being what it is, despite the isolation provided by the partitioned stack, the situation can still arise, where we might need to prevent more than one thread from accessing a global variable, or executing a given function, or other block of code. For instance, we may be in the process of assembling a linked list, and it would be counter-productive to have two threads adding an element to the end of the list at the same time.
To cater for such eventualities, the thread library comes equipped with a useful gadget called a mutex -–which is an abbreviation of ‘mutually exclusive switch’.

The above notwithstanding, here is a dire warning: if you need to use a mutex, you’ve done it wrong. The risks associated with adding a mutex, which are outlined below, together with the fact that a mutex totally negates any advantage of multi-threading, make its use highly undesirable.

We declare mutexes (mutices?) globally since, for obvious reasons, they need to be thread-visible.

```
pthread_mutex_t xmutex;
```

We then have to put in some code to lock the mutex, which warrants a more detailed discussion.

The mutex is not a magic device and, in reality, it is only the same as a flag, or semaphore. The whole question of mutex protection is analogous to that of file

locking. Everything has to be done by agreement, and all parties wishing to access the resource have to cooperate.
Accordingly, the correct way to do this, is as follows:

- As early as possible, initialise and unlock the mutex
- Just before the protected block of code, attempt to lock the mutex
- If this fails, it means that another thread got there first, so sleep and retry until it succeeds
- If it succeeds, lock the mutex and execute the block of code
- Just after the protected block of code, unlock the mutex.

This model makes several assumptions:

- It is impossible for two threads to lock the mutex simultaneously.
- It is impossible for two threads to fail to lock the mutex simultaneously.

These assumptions are correct in 99.999% of cases. Where they are incorrect, we can become introduced to the condition known, affectionately, as ‘thread deadlock’.

Down this road lies madness.

Even running the code through a debugger, to find out the cause of the deadlock, is non-trivial, as we need to switch between threads for every line of code executed.
If we persevere, both threads will arrive at the mutex, to find that there is no problem.
Debuggers can’t follow thread race conditions.

Anyway, we initialise a mutex to the unlocked condition, using the mutex_init function:

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

The variable ‘attr is a pointer to an attribute structure, which contains only one member, which is a pointer to an int. We are not interested in changing this, so we pass in NULL, to get the default mutex conditions.

```
if(pthread_mutex_init(&xmutex, NULL) == -1){
   printf(“Failed to initialise mutex\n”);
}
```

So, we now have an initialised mutex, and can protect our precious code. There are two calls available to do this. We can either

- Explicitly attempt to lock it, and check the return value:

```
if(pthread_mutex_lock(&xmutex) == -1){
   printf(“Failed to lock mutex\n”);
}
```

- We can call a function which does this for us:

```
if(pthread_mutex_trylock(&xmutex) == -1){
   printf(“Failed to lock mutex\n”);
}
```

If we want to wait for the code to become available, we will need a loop, which we traverse, just before the block of code:

```
   while(pthread_mutex_trylock(&mutex) == -1){
      sleep(1);
   }
```

Then, at the end of our protected block of code, we add the lines:

```
if(mutex_unlock(&xmutex) == -1){
   printf("Failed to unlock mutex\n");
}
```

Now we're ready to try it. using our example code, above.
We will add a mutex to the code, to force the threads to execute it in their order of arrival.
As it happens, this will not affect the overall timing, since the operation performed by each thread is sleep(). However, this generalisation does not apply to other functions. If we were performing real computationally intensive tasks, in our function, the advantages of multi-threading would be totally negated, as the threads would have to queue up to execute the function.

```
#include <stdio.h>
#include <ctype.h>
#include <pthread.h>
#define __REENTRANT

void *func(void *);
pthread_mutex_t mutex;

main(argc, argv)              /* main */

int argc;
char **argv;

{

pthread_t thr[100];
int delay;
int i;

if(argc < 2){
      delay = 10;
   } else {
      delay = atoi(argv[1]);
   }

if(pthread_mutex_init(&mutex, NULL) == -1){
      printf("Failed to initialise mutex\n");
   }

for(i = 0; i < 100; i++){
      if((pthread_create(&thr[i], NULL, func,  (void *)&delay)) != 0){
```

```
            printf("Failed to create thr[%d]\n",i);
        }
    }
    for(i = 0; i < 100; i++){
        if(pthread_join(thr[i], NULL) != 0){
            printf("Failed to start thr[%d]\n",i);
        }
    }


    printf("All threads terminated\n");

}                       /* main */

void *
func(delay)                     /* func */

void *delay;

{

    while(pthread_mutex_trylock(&mutex) == -1){
        sleep(1);
    }

    printf("Starting thread %d for %d sex..\n", pthread_self(), *(int *)delay);
    sleep(*(int *)delay);
    printf("Thread %d returning..\n", pthread_self());

    if(pthread_mutex_unlock(&mutex) == -1){
        printf("Can't unlock mutex\n");
    }

}                       /* func */
```

Performance Note

So, we go to all the trouble of parallelising our application, we create a hundred threads, with no mutexes, and sit back and watch it run.
Since we have one hundred parallel paths of execution, our process will run one hundred times faster, right?

The answer is, that it will, but our application probably will not..

The process, within its process slot, will certainly run one hundred times faster, but that’s not the whole story. The real question we should ask is, ‘How often does the process run?’

If our process is the only one on the machine, then the application will certainly complete its operations considerably faster. However, in reality, we will be competing for CPU time with several dozen other processes, and will have to take turns at running.

If our one hundred threads were one hundred processes, the overall application would, indeed, run nearly one 100 times faster. If our application is performance critical, then there is no choice, but to do it that way.

However, performance is not always the sole criterion, and we may also be concerned with architectural elegance, or overall simplicity.

There are many situations, where each thread must read and write global data within the parent process. The overhead of synchronising communication with many threads, and shunting data back and forth, may not be worth the improvement in performance.

Additionally, there may be computational reasons for using multi-threading, simply because we actually need to perform some operations in parallel. The simple example of the threaded server, where we leave the main thread listening for the next connection, while subsidiary threads handle each current connection is a good example.