SIENAcollege
Computer Science

Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2024

Topic Notes: Critical Sections with POSIX Threads

You were introduced to the basics of *POSIX Threads* or *pthreads* in our current lab.

The basic idea is that we can create and destroy threads of execution in a program, on the fly, during its execution. These threads can then be executed in parallel by the operating system scheduler. If we have multiple processors, we should be able to achieve a speedup over the single-threaded equivalent.

In the lab, you saw how to do perform the basics:

- `pthread_create` – Creates a new thread, stores a thread id that can be used to refer to this thread later in the first parameter, and runs the function specified as the third parameter in a new thread.
- `pthread_exit` – Causes the calling thread to exit. This is called implicitly if the thread function called during the thread creation returns. Its argument is a return status value, which can be retrieved by `pthread_join()`.
- `pthread_join` – Causes the calling thread to block (wait) until the thread with the identifier passed as the first argument to `pthread_join` has exited, can optionally pass a second argument as a pointer to a location where the return status passed to `pthread_exit()` can be stored.

Brief Review of Critical Sections

A great thing about using threads instead of message passing is that different threads can share memory, meaning when one thread needs information that another is "responsible" for, it can usually just access that information.

However, as you have seen in other contexts, concurrent access to shared variables can be dangerous.

To see this, we will look at a series of examples in

`https://github.com/SienaCSISParallelProcessing/pthread-mutex`

We will first look at the program in `pthread_danger`.

Before we look at the potential problems here, notice that there is a bit of extra initialization in these examples. This is necessary in some pthreads implementations to make sure the system will allow your threads to make use of all available processors. It may, by default, allow only one thread in your program to be executing at any given time. If your program will create up to n concurrent threads, you can make the call:

```
pthread_setconcurrency(n+1);
```

somewhere before your first thread creation. The "+1" is needed to account for the original thread plus the n you plan to create.

You may also want to specify actual attributes as the second argument to `pthread_create()`. To do this, declare a variable for the attributes:

```
pthread_attr_t attr;
```

and initialize it with:

```
pthread_attr_init(&attr);
```

and set parameters on the attributes with calls such as:

```
pthread_attr_setscope(&attr, PTHREAD_SCOPE_PROCESS);
```

Then, you can pass in `&attr` as the second parameter to `pthread_create()`.

For the FreeBSD and Linux implementations we are using, the defaults appear to work well, so you can safely leave these extra steps out.

As for the main "functionality" here, we can see that the program creates whatever number of threads are specified by the program's command-line parameter. Each of those threads increments a shared global variable `counter` 100,000 times.

Run it with one thread, and we get 100000. What if we run it with 2 threads? On a multiprocessor, it is going to give the wrong answer! Why?

The answer is that we have concurrent access to the shared variable `counter`. Suppose that two threads are each about to execute `counter++`, what can go wrong?

`counter++` really requires three machine instructions: (*i*) load a register with the value of `counter`'s memory location, (*ii*) increment the register, and (*iii*) store the register value back in `counter`'s memory location. Even on a single processor, the operating system could switch the process out in the middle of this. With multiple processors, the statements really could be happening concurrently.

Consider two threads running the statements that modify `counter`:

Thread A		Thread B	
A_1	`R0 = counter;`	B_1	`R1 = counter;`
A_2	`R0 = R0 + 1;`	B_2	`R1 = R1 + 1;`
A_3	`counter = R0;`	B_3	`counter = R1;`

Consider one possible ordering: A_1 A_2 B_1 A_3 B_2 B_3 , where `counter=17` before starting. Uh oh.

What we have here is a *race condition* that can lead to *interference* of the actions of one thread with another. We need to make sure that when one process starts modifying `counter`, that it finishes before the other can try to modify it. This requires *synchronization* of the processes.

If we run it on a single-processor system (do they make those anymore?), the problem is much less likely to show itself - we will probably get the correct result when we run it. However, there is no guarantee that this would be the case. The operating system could switch threads in the middle of the load-increment-store, resulting in a race condition and an incorrect result.

We need to make those statements that increment `counter` *atomic*. We say that the modification of `counter` is a *critical section*.

There are many solutions to the critical section problem and this is a major topic in an operating systems course. But for our purposes, at least for now, it is sufficient to recognize the problem, and use available tools to deal with it.

Mutual Exclusion with pthreads

The pthread library provides a construct called a *mutex* (short for the *mutual exclusion* that we want to enforce for the access of the `counter` variable) allows us to ensure that only one thread at a time is executing a particular block of code. We can use it to fix our "danger" program:

`pthread_nodanger`

We declare a mutex like any other shared variable. It is of type `pthread_mutex_t`. Four functions are used:

- `pthread_mutex_init` – Initialize the mutex and set it to the "unlocked" state.
- `pthread_mutex_lock` – Request the lock on the mutex. If the mutex is unlocked, the calling thread acquires the lock and proceeds. Otherwise, the thread is blocked until the thread that previously locked the mutex unlocks it and the calling thread obtains the lock.
- `pthread_mutex_unlock` – Unlock the mutex. If any other threads are blocked on this mutex in a call to `pthread_mutex_lock`, one of them obtains the lock and can proceed past its `pthread_mutex_lock` call.
- `pthread_mutex_destroy` – Destroy the mutex (clean up memory).

A few things to consider about this:

Why isn't the access to the mutex a problem? Isn't it just a shared variable itself? – Yes, it's a shared variable, but access to it is only through the pthread API. Techniques that are discussed in detail in an operating systems course are used to ensure that access to the mutex itself does not cause a race condition.

Doesn't that lock/unlock have a significant cost? – Let's see. We can time the programs we've been looking at:

`pthread_danger_timed`

`pthread_nodanger_timed`

Try these out. What are the running times of each version? Perhaps the cost is too much if we're going to lock and unlock that much. Maybe we shouldn't do so much locking and unlocking. In this case, we're pretty much just going to lock again as soon as we can jump back around through the `for` loop again.

This is a good example of the parallel overhead we mentioned earlier when talking about the limitations on efficiency and scalability.

Here's an alternative:

`pthread_nodanger_coarse`

In this case, the coarse-grained locking (one thread gets and holds the lock for a long time) should improve the performance significantly. How fast does it run now? But at what cost? We've completely serialized the computation! Only one thread can actually be doing something at a time, so we can't take advantage of multiple processors. If the "computation" was something more significant, we would need to be more careful about the granularity of the locking.

Barrier Synchronization with pthreads

As we have seen with MPI and the `MPI_Barrier` function, parallel programs often need to have their cooperating tasks synchronize at a given execution point.

The idea is straightforward, but it's a bit different than the MPI approach.

- Declare a structure of type `pthread_barrier_t` to manage the barrier.

  ```
  pthread_barrier_t barrier;
  ```

- Initialize the barrier:

  ```
  pthread_barrier_init(&barrier, NULL, num_threads);
  ```

  Here, the third parameter indicates the number of threads that need to enter the barrier before all threads are released to continue execution.

- To use the barrier, all threads must call:

  ```
  pthread_barrier_wait(&barrier);
  ```

  All threads that call this function will remain in the function until the total number of threads that were specified in the `pthread_barrier_init` call have entered the barrier, at which time all threads will be released to continue, and the barrier is reset.

- Destroy the barrier when it is no longer needed:

  ```
  pthread_barrier_destroy(&barrier);
  ```