CS 220: Introduction to Parallel Computing

Critical Sections

Lecture 23

Process Ordering

- You may have noticed that when we print to the terminal, the order changes for every run
  - True for both MPI and pthreads
- This happens for a couple of reasons:
  - We have no control over the actual execution of threads or processes
    - Controlled by the OS **scheduler**
  - The terminal only accepts one line at a time from a process (this is why we don't get jumbled output)

The Scheduler (1/2)

- The simplest form of scheduling is “round robin”
  - Go around in a loop and give everybody a little time
- In reality, operating systems generally use **priority queues** and more advanced logic to choose how to run our threads
- Some threads may be a higher priority than others, some may be waiting for I/O to complete, etc…

The Scheduler (2/2)

- If your computer has multiple CPUs or multiple cores, then the scheduler decides which cores run your processes
- If you launch 1000 threads, then the scheduler tries to give them all a fair share of the CPU
  - Resource allocation
- The main thing to remember: we don't have direct control over how the scheduler chooses to run our threads

Global Variables

- Let's take a look at what happens when multiple threads access a global variable at the same time
- Be **very** careful with globals!
  - For example: let's assume you write a program with global variable **i**
  - Later, in a thread, you want to iterate through some values and forget to declare a local **i**

Race Condition

- When multiple threads have access to a variable, **race conditions** can occur
- This happens when two threads “race” to read/write a value in memory
  - The sequence of events is not controlled
- Thread 1 wants to subtract 10 from variable A
- Thread 2 wants to add 2 to variable A
- Which happens first? What will be the outcome?

Example

- We have two threads, A and B
- A and B both want to add 1 to a shared variable, count
- What are the different scenarios that can play out here?
- What happens if we don’t call pthread_join on the threads?

Handling Race Conditions

- In general, race conditions are not desirable!
  - Having your code do unpredictable things is almost always bad
- We want to have control on how events unfold
- In other words, we wish to **serialize** some portions of our programs
- We can do this with **critical sections**

Critical Section

- A **critical section** is a block of code that is protected from concurrent access
- We set up a particular region of our code and then only allow a single thread to access it at a time
- How can we implement critical sections?

Busy Waiting

- One approach for creating critical sections in our code is called **busy waiting**
- Wait for your turn in a while loop
  - ```
    while (turn != my_thread_id) {
        /* Wait … */
    }
    ```
- Once it's your turn, enter the critical section, do your work, and then set "turn" to the next thread when you're done
    ```

Busy Waiting: Downsides

- The problem with busy waiting is that the threads are constantly checking for their turn
  - Your CPU will spike up to 100% usage as the thread continues to check, and check, and check…
- There isn't much of a speed improvement over a serial program because so much wasted work is taking place!
- There **has** to be a better way…

Mutex

- In parallel programming a **mutex** ensures that only one thread can enter a critical section at a time
- Mutex: *Mutual Exclusion*
- This lets you "lock" part of your code so that other threads cannot access it
- We don't have the concept of a mutex in MPI… why not?

Using a Mutex

- To create a mutex, use:
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Note the type: pthread_mutex_t
- Now let's use the mutex to protect our code:

```
pthread_mutex_lock(&mutex);
shared_var = shared_var + 1;
pthread_mutex_unlock(&mutex);
```

Notes

- There are other ways to define a critical section
- We'll be going through several parallelism primitives over the next few class periods
- Shared variables don't **have** to be globals
  - You can allocate memory and pass a pointer to your threads

Try it Out

- Let's make sure you can run a basic pthreads application
- Create some threads and have them modify a global variable all at once
- Then protect access to the variable with a mutex
- Question: are we benefitting from parallelism here?