CS 521: Systems Programming

Critical Sections

Lecture 13

Today's Schedule

- Critical Sections and Busy Waiting
- Mutexes
- Barriers

Today's Schedule

- **Critical Sections and Busy Waiting**
- Mutexes
- Barriers

Process/Thread Scheduling

- You may have noticed that when we print to the terminal in a multi-threaded application, the order changes for every run
- 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 CPU 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
  - *multi-level feedback queues*
- Some threads may be a higher priority than others, some may be waiting for I/O to complete, etc…

The CPU 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
- 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!

Race Conditions

- 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

- 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…

Today's Schedule

- Critical Sections and Busy Waiting
- **Mutexes**
- Barriers

Mutex

- In parallel programming a **mutex lock** ensures that only one thread can enter a critical section at a time
  - Mutex: *Mutual Exclusion*
  - Also sometimes just called a *lock*
- This lets you "lock" part of your code so that other threads cannot access it
  - (temporarily)

Using Mutexes

- 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);`

Mutex Declaration

- Where you declare your mutex is very important
- For example, what happens when each thread creates its own mutex?
  - This is basically like checking if you have the keys to your own house
    - (you always do… right?)
- In general, mutexes should be a shared resource
  - Declared globally

Mutexes: Mental Model [1/2]

- You can think of a mutex as a protector of a shared resource that only one thread can access at a time
- It's the gatekeeper for your protected resource
- You'll almost always have:
  - The mutex
  - The variable you're protecting

Mutexes: Mental Model [2/2]

- Let’s say our shared resource is the whiteboard
- Before you can write on the whiteboard, you have to ask the instructor first
- The instructor will only allow one student to write on the board at a time
    - …if you request to use the whiteboard while someone else is already using it, then the instructor makes you wait

Checking a Mutex

- What happens when we try to lock a mutex that is already locked by another thread?
  - We block!
- In some cases, we want to determine whether we can lock the mutex, but move on if we cannot:
  - `pthread_mutex_trylock(&mutex)`
- Even if the mutex is already locked by another thread, the function call returns immediately

Some Notes

- There are other ways to define a critical section
- We'll be going through several parallelism primitives in class
- Shared variables don't **have** to be globals
  - You can allocate memory (via `malloc` etc) and pass a pointer to your threads

Today's Schedule

- Critical Sections and Busy Waiting
- Mutexes
- **Barriers**

Syncing Up

- Sometimes we want to synchronize all our threads
  - Say, we want them all to compute a particular value or call a function before starting their work in parallel
- We can use a **barrier** to ensure all the participating threads call a particular function before moving on
- `pthread_barrier_init(pthread_barrier_t *bar_p, N unsigned count);`
  - why is `count` important here?
- `pthread_barrier_wait(pthread_barrier_t *bar_p);`
- `pthread_barrier_destroy(pthread_barrier_t *bar_p);`

Barrier Issues

- Not all implementations of pthreads support barriers
  - In particular, macOS does not include them
- Using many barriers can reduce performance – you'll only be able to move past the barrier when the **slowest** thread gets to it!
- Prefer approaches that require less synchronization and coordination for best performance