Condition Variables and Semaphores

CSC 213 – February 25, 2021
Agenda for Today

1. Bug of the Day
2. Assignment Q&A
3. Waiting in a Parallel Program
4. Condition Variables
5. Semaphores
6. Wrap Up
Bug of the Day
What's wrong here?

```c
pthread_t thread1;
int thread_arg = 1;
if (pthread_create(&thread1, NULL, thread_fn, &thread_arg))
    exit(2);

pthread_t thread2;
thread_arg = 2;  // this might happen before thread 1 reads its argument.
if (pthread_create(&thread2, NULL, thread_fn, &thread_arg))
    exit(2);

...  // Assuming (incorrectly) that thread 1 must start
    // when we create it, and will read thread_arg before
    // its changed to 2. Order Violation.
```
Q&A
Are we allowed to use a spinlock for the critical section in our code?

If you mean “should I write my own lock implementation” the answer is no.

If you’re asking “should I write something like this:”
while (not holding lock) { acquire lock }

You may have a misunderstanding about the pthread_mutex interface. When you call pthread_mutex_lock, it only returns when you’ve acquired the lock. Normally, you guard a critical section with a mutex like this:

```c
pthread_mutex_lock(&m);
// Do critical work here
pthread_mutex_unlock(&m);
```

There’s no extra loop associated with the locking, although you’ll definitely need a loop to traverse the input array in this assignment.
You said the thread functions should be “doing the counting” but what does that mean?
My intention was that you would move the code to count letters into your thread function. Your count_letters function will be almost entirely replaced by code to divide up the input and create threads to do the counting.

How should we divide up input in the count_letters function?
Let’s assume we have four threads for now.
One option is to split input array into four contiguous sections. So thread 0 will count from index 0 to index (size / 4). Thread 1 will pick up at the next index, and so on. You may have a little extra input to give to the last thread if the input size doesn’t divide evenly by 4

Another option is to make the threads count non-contiguous ranges. So, thread 0 might count input characters 0, 4, 8, 12, 16, etc. Thread 1 would count characters 1, 5, 8, 13, ...

As long as every character is counted exactly one time, and all the threads do roughly-equal amounts of counting, your approach is fine.
Letter Count Assignment Q&A
Waiting in a Parallel Program
A Simple Parallel Program

Main Function
Create two threads, tick_thread and tock_thread

While there is input from the user:
  If the user typed "go", allow both threads to run
  Wait for more input.

Join with tick_thread and tock_thread

Thread Function
Wait for the main thread to allow this thread to run.

Repeat five times:
  Print this thread's message ("tick" or "tock")
  Pause for one second

How should we make the threads wait, and how do we resume them?
Discuss strategies with your group. We'll complete the implementation together.
A Simple Parallel Program

Approach #0: Use a Lock
Threads will spin: acquire a lock, check if `go == true`, then release lock. Try again if `go` was false
Main thread will acquire the lock, set `go` to true, then release lock.
Why not use this approach? The threads are busy waiting, so we’re tying up a processor to do no work. The waiting threads are competing with the main thread for the lock. The main thread may have to wait a very long time to acquire the lock.

Approach #1: Use a Condition Variable
Threads will call pthread_cond_wait to sleep
Main thread uses pthread_cond_signal to wake threads
Global state: a condition variable, a mutex, and a boolean to track whether threads can run.

Approach #2: Use a Semaphore
How do we use this? We’ll return to this one.
A Simple Parallel Program
Condition Variables
Condition Variable Basics

What does a condition variable do?
Makes it possible for a thread to wait for a condition to become true.

How does the condition variable know what condition you’re waiting for?
It doesn’t. We actually just use the condition variable for waiting, and check the condition in the code around it. If a thread changes program state and that change might satisfy the waiting condition for another thread, it should wake it up with a condition variable.

Why would we use a condition variable with a mutex instead of just the mutex?
When we use a lock alone, threads have to constantly check for state changes that would allow them to make progress. A condition variable lets us put threads to sleep when a condition is false, and wake them up when we need them to check the condition again.
Condition Variable Interface

`pthread_cond_wait(pthread_cond_t* cv, pthread_mutex_t* m)`

Put the calling thread to sleep. The calling thread must hold mutex m. When the thread sleeps, `pthread_cond_wait` will unlock m. When the thread wakes back up, this function re-acquires m and then returns.

This function might wake up the thread and return even if it was never signaled.

`pthread_cond_signal(pthread_cond_t* cv)`

Wake up a thread sleeping on this condition variable. If there are multiple threads sleeping on the condition variable, this is only guaranteed to wake one up.

`pthread_cond_broadcast(pthread_cond_t* cv)`

Wake up all threads sleeping on this condition variable.
Implement Tick/Tock using Condition Variables
Semaphores
Semaphore Basics

What is a semaphore for?
A semaphore can be used like a lock, and for ordering events in a parallel program. Semaphores have a value, and work a bit like a counter. Threads can wait by decrementing the counter below zero. A thread can wake up waiters by incrementing the counter.

How is a semaphore different from a mutex or condition variable?
Unlike a condition variable, a semaphore has an integer value that remembers the number of waiters or wakeups. Using a semaphore like a condition variable makes it impossible for there to be lost wakeups.

Unlike a mutex, we could set up a semaphore so three, four, five, etc. threads can acquire it at a time. This may help us limit the number of threads using some shared resource, even if we want to allow a small number of threads and not just one to use it at a time.
Semaphore Interface

sem_init(sem_t* s, int cross_process, int state)
Initialize a semaphore. The `s` parameter points to the semaphore being initialized. The `cross_process` parameter is usually zero, unless you’re using a semaphore across processes. The `state` parameter is the starting value of the counter in the semaphore.

sem_wait(sem_t* s)
Decrement the semaphore’s counter.
If the counter is negative after decrementing it, the thread will wait.
If the counter is greater than or equal to zero, the thread can continue immediately.

sem_post(sem_t* s)
Increment the semaphore’s counter.
If there is at least one thread waiting on the semaphore, wake one up.
Implement Tick/Tock using Semaphores
Wrap Up
Reminders

Parallel Lettercount Assignment
Due today at 11:59pm

Lab Tomorrow
Read through the lab if you haven't already.
You can work on your own if you'd like. I'll post instructions right now.

Are you declaring a CS major this spring?
Attend today's CS Extra at 5pm to learn the process. A recording will be available if you can't make it.