CS 241: Synchronization

This week, we are going to be building synchronization primitives and using mutexes in order to implement some basic data structures.

Warm-Up Questions

What is a critical section? How can we protect a critical section?

How do C mutexes work with shared variables? Does each mutex know what data it's protecting?

What is a condition variable? Why do we need one? Why should we wait on condition variables in a loop?

What is a semaphore? What methods may block? What methods do not block? What is a binary semaphore? (For a binary semaphore that starts at 1, always `sem_wait(...)` before `sem_post(...)`.)

The Ambitious Thread

The *ABA problem* is a very tricky problem in concurrent programming. Reusable barriers aren't inherently the same thing, but pseudo-ABA problems go something like this:

- Thread #1 reads memory address x and gets the value A
- Thread #1 gets stopped (preempted) and Thread #2 starts running
- Thread #2 sets x to B, and a while later, back to A
- Thread #1 resumes running, reads x, and gets A again
- Thread #1 thinks x hasn't changed, even though it has!

So, that leads to the following question: why can't we implement a reusable `barrier_wait` like this?

```
pthread_mutex_lock(&m);
remain--;
if (remain == 0) {
    pthread_cond_broadcast(&cv);
    remain = num_threads;
}
else {
    while(remain != 0) {
        pthread_cond_wait(&cv, &m);
    }
}
pthread_mutex_unlock(&m);
```

Try to give as much detail as possible. Multi-threaded programming is hard, so describing the problem in as much depth and detail on paper will prevent race conditions.

Algorithm Design

Before you write your queue or semamore, write out the steps. Create a list of every check/function call you make.

```
void semm_post(semm_t *sem)
```

- Check if the semaphore ptr is null (not entirely necessary)
- Increment the semaphore count
- If semaphore count is __, I should ...

```
void semm_wait(semm_t *sem)
```

```
void queue_push(queue_t *que)
```

```
void *queue_pull(queue_t *que)
```

Thread-Safe Queue

In multithreaded code, there is a strong notion of ownership when it comes to memory and information. What would be the problem with implementing `int queue_size(...)`? How about `void* queue_peek(...)`? How might we otherwise tell that the queue is empty? (Hint: How do you know that a C string is over?)