CS241 - Synchronization

This week you 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 wrap it up in a loop?

What is a semaphore? What methods may block? What methods do not block? What is a binary semaphore? (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 does something
- Thread #1 gets stopped (pre-empted), Thread #2 Runs for a long time and ruins Thread #1 Data structure
- Thread #2 gets stopped and Thread #1 can continue
- But Thread #1 Data is corrupt!

So that leads to the question, Why can't we just have our barrier wait be this one from the wikibook?

```
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 go over and write up your queue/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. Think of a problem if we implemented `int queue_size(...)`, how about `void* queue_peek(...)`? How can we otherwise tell that the queue is empty? (Hint: How do you know that a C string is Over)