

# CSCI 251: Concepts of Parallel and Distributed Systems

Alvin Lin

September 20th, 2017

## Topics

- Race conditions and mutexes
- Overheads
- Load balancing

## Race Conditions

Suppose we are finding the number of primes in a given range. If we are checking if a number  $n$  is prime, we can check for factors up to  $\sqrt{n}$  as a simple optimization. If we have a function `isPrime(n)` to be distributed across 2 threads, we can divide  $n$  into two ranges for checking, one range being 3 to  $\frac{n}{2}$ , and the second range being  $\frac{n}{2}$  to  $n$ .

```
#include <pthread.h>

pthread_t t1,t1;
struct prime_arg t1arg = {3,n/2};
struct prime_arg t2arg = {n/2+1,n};
pthread_create(&t1, NULL, prime_thread, &t1arg);
pthread_create(&t2, NULL, prime_thread, &t2arg);
...
pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

Suppose these threads increment a `pcount` variable to count primes, a race condition can occur if two threads are trying to read and write to a single piece of memory at

the same time (namely, the `pcount` variable). Race conditions cause unpredictable behavior.

## Mutex Locks

A mutex lock ensures that two threads cannot simultaneously attempt to update the `pcount` variable. When a thread acquires a lock, it ensures that operation to be performed is atomic.

```
p_thread_mutex_lock(&mutex);  
pcount++;  
p_thread_mutex_unlock(&mutex);
```

This is a problem however, since the mutex becomes a bottleneck for the threads. We can remove the need for a mutex lock by having each thread maintain a local count and accumulating at the end.

## Load Balancing

In the previous example, dividing the input range  $n$  among two threads is a naive solution since one thread may finish first. This solution is not load balanced since one thread will be idle at the end of its set of operations.

## Programming Guidelines

### Memory

Every memory location in your program must meet at least one of the following criteria:

- Only one thread can access it.
- It is immutable.
- It is synchronized (locks are used to avoid race conditions).

### Synchronization

- Avoid data races, use locks to prevent it.
- Use consistent locking, each shared location that needs synchronization should have a corresponding lock.

- Coarse grained vs fine grained locks, you can lock memory locations individually or lock the entire block, depending on your needs.
- Critical sections, the cost of the critical section is given by the time to execute the critical section and the resources used by the OS.

## Reminders

The midterm is on October 11th. Refer to MyCourses for details on Project 1.

Professor Mohan Kumar:  
mjkvcs@rit.edu  
<https://cs.rit.edu/~mjk>

Rahul Dashora (TA):  
rd5476@mail.rit.edu

You can find all my notes at <http://omgimanerd.tech/notes>. If you have any questions, comments, or concerns, please contact me at [alvin@omgimanerd.tech](mailto:alvin@omgimanerd.tech)