# CMPS270 Assignment 3 Report

**Hardware Overview:**

Model Name: MacBook Pro

Model Identifier: MacBookPro15,4

Processor Name: Quad-Core Intel Core i5

Processor Speed: 1.4 GHz

Number of Processors: 1

Total Number of Cores: 4

L2 Cache (per Core): 256 KB

L3 Cache: 6 MB

Hyper-Threading Technology: Enabled

Memory: 8 GB

System Firmware Version: 1731.140.2.0.0 (iBridge: 19.16.16067.0.0,0)

OS Loader Version: 540.120.3~22

Serial Number (system): FVFC20Q2L410

Hardware UUID: B1A5AFF2-8A5D-52E1-88CD-C1C1A5D61A59

Provisioning UDID: B1A5AFF2-8A5D-52E1-88CD-C1C1A5D61A59

Activation Lock Status: Enabled

## Count Race

When a device or system tries to do two or more operations simultaneously when, due to the nature of the device or system, the actions must be performed in the correct order to be performed successfully, a race condition results.

The most frequent associations of race with programming and computer science are these. They happen when two processes, or threads, in a computer program attempt to access the same resource concurrently and disrupt the system.

A frequent problem for multithreaded programs is race circumstances.

A simple example of a race condition is a light switch. In some homes, there are multiple light switches connected to a common ceiling light. When these types of circuits are used, the switch position becomes irrelevant. If the light is on, moving either switch from its current position turns the light off. Similarly, if the light is off, then moving either switch from its current position turns the light on.

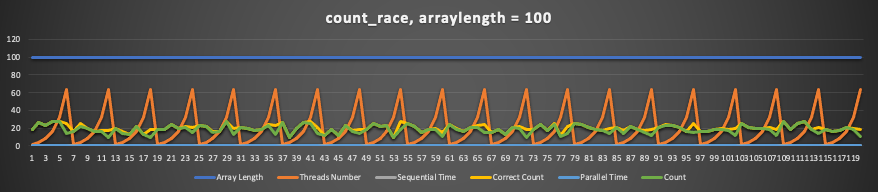
With that in mind, imagine what might happen if two people tried to turn on the light using two different switches at the same time. One instruction might cancel the other or the two actions might trip the circuit breaker.

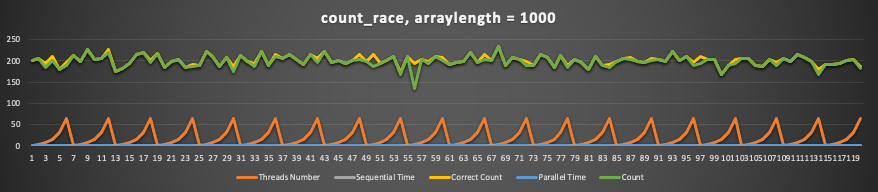
A simple example of a race condition is a light switch. In some homes, there are multiple light switches connected to a common ceiling light. When these types of circuits are used, the switch position becomes irrelevant. If the light is on, moving either switch from its current position turns the light off. Similarly, if the light is off, then moving either switch from its current position turns the light on.

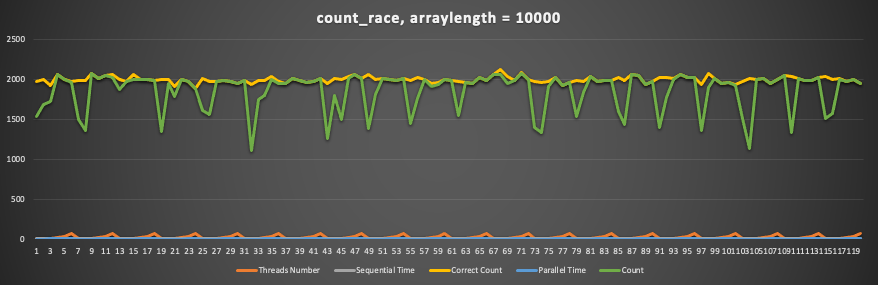
With that in mind, imagine what might happen if two people tried to turn on the light using two different switches at the same time. One instruction might cancel the other or the two actions might trip the circuit breaker.

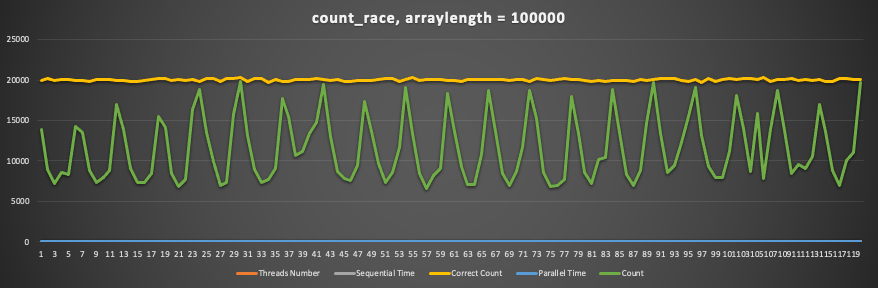
A race condition may happen in computer memory or storage if requests to read and write a significant amount of data are almost simultaneously received and the machine tries toWhilst the old data is being read, overwrite some or all of it. The outcome might be any of the following:

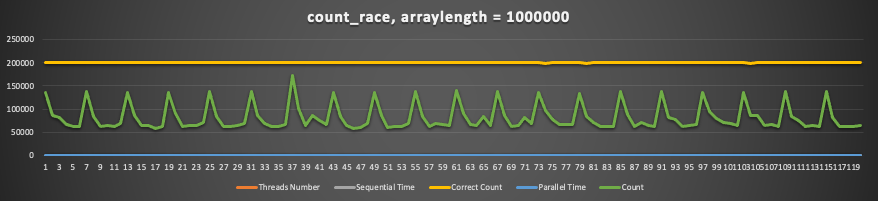
* the application is found to be operating illegally, or the machine crashes
* errors reading the old data
* errors writing the new data

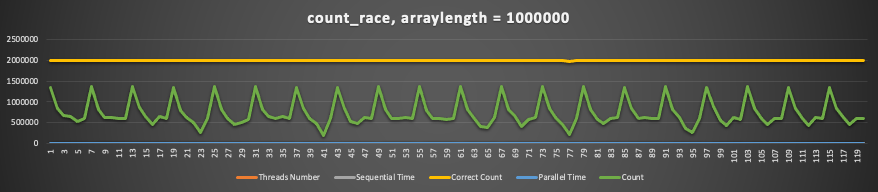












## Mutex

A program is a set of instructions being executed and multiple threads of a program can be executing different sections of the program code. However, caution should be exercised when threads of the same process attempt to simultaneously execute the same portion of code.

Critical Section:

Critical section is any piece of code that has the possibility of being executed concurrently by more than one thread of the application and exposes any shared data or resources used by the application for access.

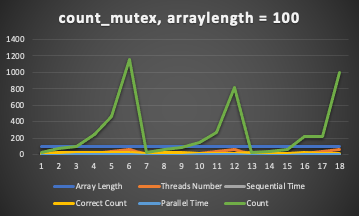
Race Condition:

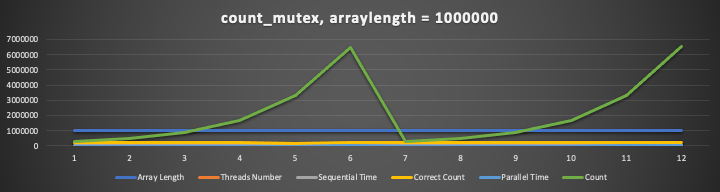
Race conditions happen when threads run through critical sections without thread synchronization. The threads "race" through the critical section to write or read shared resources and depending on the order in which threads finish the "race", the program output changes. In a race condition, threads access shared resources or program variables that might be worked on by other threads at the same time causing the application data to be inconsistent.

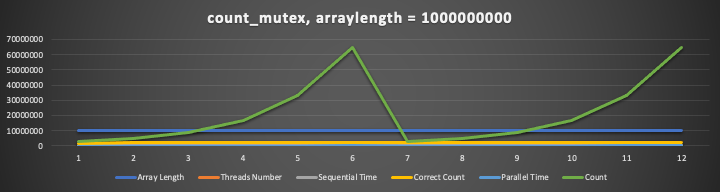
As an example, consider a thread that tests for a state/condition, called a predicate, and then takes subsequent action based on that condition. This sequence is called test-then-act. The pitfall here is that the state can be mutated by the second thread just after the test by the first thread and before the first thread acts based on the test. A different thread changes the predicate in between the test and act. In this case, action by the first thread is not justified since the predicate doesn't hold when the action is executed.

Example Thread Race

The program below spawns two threads. One thread prints the value of a shared variable whenever the shared variable is divisible by 5. A race condition happens when the printer thread executes a test-then-act if clause, which checks if the shared variable is divisible by 5, but before the thread can print the variable out, its value is changed by the modifier thread. Some of the printed values aren't divisible by 5, which verifies the existence of a race condition in the code







## Count Private

Here, threads number seem to not bear a significant effect on the accuracy of the program, as it seems to be overwhelmingly different from the correct count.

There is no performance improvement because the threads are not synchronized.

Private variables are not shared across threads, so each thread has its own copy. This means that each thread can update its own copy without having to worry about race conditions. However, because the threads are not synchronized, they may not all be updating the same copy of the variable. This can lead to inconsistency in the results.