Race Conditions

A *race condition* is a situation where the outcome of a program depends on the exact state of the system in a way that is not predictable and often varies each time the program is run.

One of the most common race conditions is called time-of-check to time-of-use (TOCTOU). This is where you check to see if a condition is true before doing things, but get distracted in the process. Specifically, you check the condition, get distracted, and then do the thing, and in the time you were distracted, the condition changed. You don't realize the condition has changed and proceed to do the action even though the condition has changed.

In code, this often happens in a situation like below:

```
if some condition is true:
    do something
```

The "time of check" is the first line, and the "time of use" is the second. If there is a delay in between those two lines, it's possible that the condition could change during that delay. The program doesn't recheck that condition, and then we end up doing the action when we shouldn't. Most of the time, nothing will happen in between checking and using, but every once in a while, often at unpredictable times, something will happen. This makes race conditions difficult to debug, as the conditions that caused the problem can be hard to reproduce.

Example 1 Suppose you are writing code for a ticket-selling site. When someone clicks on an event, you first check to see if there are any tickets available. If you don't check again before the person clicks the buy button, then it's possible all the remaining tickets were bought in the time between when the first click happened and when the user clicked the buy button. This could end with your code selling the same tickets to two people or overselling the event.

Example 2 Suppose we are running a course registration system. The system can't handle more than 10 concurrent students. The following code is used to log students onto the system:

```
if get_num_users() < 10:
    add_user(student)
else:
    print('Come back later.')
```

Here is where the TOCTOU race condition can come into play: The if statement runs to check if the number of users is less than 10. Right after that, a context switch happens. After a while, another context switch happens and the program picks up with the `add_user` line. But what if in the meantime other students logged in and pushed the number of users up to 10? The program would have no way of knowing that. After the context switch, it was in suspended animation and the last it knew there were less than 10 students. When it resumes, it picks up right where it left off and has no idea that things have changed in the interim.

We could also get a problem if two threads are running this code each on different CPU cores. If both threads end up checking the number of users at roughly the same time, then they will both conclude there are less than 10 users. If there are 9 users on the system, we will end up with 11.

This race condition could be fixed using locks or semaphores (which will be covered in a later set of notes).

Example 3 Suppose two people, Alice and Bob, share a bank account with $1000 in it. They are currently each at ATMs in different cities, with Alice trying to withdraw $700 and Bob trying to withdraw $500. Suppose the ATM runs code that looks like this:

```
if withdraw_amount >= get_account_balance():
    process_withdrawal()
```

It's possible that the ATMs that Bob and Alice are at each run the if statement at nearly the same time. In both cases, the account balance is $1000, so both of their withdrawals will pass the if statement. If no further checks of the account balance are made while processing the withdrawal, then Alice will get $700 and Bob will get $500, which total more than is actually in the account. Depending on how the rest of it is programmed, it's quite possible that only one of the withdrawals will be reflected in the account balance. Similar vulnerabilities to this have been exploited on real ATMs in the past. One solution would be a lock that only allows one transaction on an account at a time.

Example 4 Here is a short threaded program that involves a bunch of threads all trying to print to the screen at once.

```
from threading import *
from time import sleep
from random import uniform

def f(num):
    for i in range(5):
        print(f'Hello from thread {num}!')
        sleep(uniform(.1,.5))

threads = []
for i in range(10):
    threads.append(Thread(target=f, args = [i]))

for i in range(10):
    threads[i].start()
```

Here is part of the output from one time I ran the program:

```
Hello from thread 0!Hello from thread 4!Hello from thread 1!Hello from thread 2!
Hello from thread 5!Hello from thread 3!Hello from thread 6!Hello from thread 7!
Hello from thread 8!Hello from thread 9!









Hello from thread 5!Hello from thread 8!

Hello from thread 4!Hello from thread 0!

Hello from thread 3!
Hello from thread 6!Hello from thread 7!
```

The output should have had one hello statement on each line, not all jumbled up like this. The problem is due to a race condition with the print statement which ends up messing with the part of the print that advances to the next line.

To fix the problem, we can change the `f` function by using a lock around the print statement:

```
def f(num):
    for i in range(5):
        print_lock.acquire()
        print(f'Hello from thread {num}!')
        print_lock.release()
        sleep(randint(1,5)/10)
```

```
print_lock = Lock()
```

Example 5 In the previous example, the screen is a shared resource that multiple threads have access to. Other examples of shared resources include lists, files, and databases. All of these may need to be protected by locks or other mechanisms. Here is an example with files.

One simple way to keep track of the number of visitors to a web site is to use a text file that stores a counter. Each time a page is viewed, code is run on the server side that reads the counter from the file, increments it, and stores the new value to the file. On quiet sites, this works pretty well. But on busy sites, this can lead to a problem. Web servers usually process requests in their own thread. So there could be multiple threads each trying to access the file at the same time. Something very similar to the `count += 1` problem from a previous section of these notes can happen here where two threads load the value of the counter at roughly the same time and only one of the counts gets stored. A worse scenario is if two threads are writing the new count to the file at the same time and their outputs end up jumbled together. For instance, if one is writing 4001 and the other is writing 4001 as well, it could come out as 40014001.

One solution would be a lock on the file. Another solution would be to use databases, which are designed to deal with these sorts of situations.

Example 6 Here is an example of a race condition with multiple threads working with a shared list.

```
from threading import *
from time import sleep
from random import *

L = []

def f():
    while True:
        if L == []:
            L.append(1)
        print(len(L))
        sleep(uniform(0,.1))

def g():
    while True:
        if L != []:
            sleep(uniform(0,.1))
            L.pop()

t1 = Thread(target = f)
t2 = Thread(target = g)
t3 = Thread(target = g)
t4 = Thread(target = g)

t1.start()
t2.start()
t3.start()
t4.start()
```

The `t1` thread periodically checks the list to see if it's empty, and adds an item to the list if it is empty. The other threads all run copies of the code in the g function. They each check to see if the list has anything in it. If it does, they remove an item from it. A sleep statement has been added in between the time of check and time of use to make the race condition more likely to happen. It can still happen without the sleep, but it's more rare.

The problem that happens here is right after checking if the list has something in it, it's possible that a context switch can happen (and the sleep actually guarantees it). Another thread can run, see that the list has something in it, pop that item, and leave the list empty. When we context switch back to the other thread, it picks up where

it left off, which was to pop from the list. However, at this point, the list is empty, and we get an error.

Example 7 Suppose we have the following code:

```
if has_permission(user, file):
    write to the file
```

The if statement checks if the user has permission to open the file, and if so, then we write to the file. In the short time between when the permission is checked and the file is written, it's possible some other process could change something in the file system so that the file object the program is working with is now pointing to a different file. A malicious user could use this to get the program to write to a file of their choice instead of the file the program was supposed to write to. This is a common attack technique on Unix systems.

Example 8 *Real-World Bug Hunting* by Peter Yaworski has a nice chapter on race conditions. One example he covers involves a site where you could only join if you got sent a link. That link was a URL that contained a unique token. When you followed that link, the website would pull the token from the URL, check that it was in their database, then register you onto the site, and finally mark the token in the database as used so that it couldn't be reused. If multiple users used the same URL at the same time, a race condition could allow all of them to join using that same URL. The problem is that the system didn't mark the token as used until after registering users. So multiple users could pass the token checking part of the code and be registered before the token was invalidated.

Example 9 (Priority inversion) Suppose we have two threads with different scheduling priorities, with Thread 1 having a higher priority than Thread 2, such that the OS scheduler will always choose to run Thread 1 instead of Thread 2 if they are both available. Suppose also that they both have access to the same lock and it's a spin lock.

Here's how a problem can occur. Let's say Thread 1 is running and then gives up the CPU for some I/O. Then Thread 2 gets to run and needs the lock. It acquires it, and then Thread 1's I/O is done, so the OS context switches back to Thread 1. If Thread 1 needs the lock at that point, then we have a problem. It needs Thread 2 to give up the lock, but Thread 2 can't do that because it isn't able to run as long as Thread 1 is running.

So we see where the name comes from: a lower priority process is preventing a higher priority process from getting work done. Something like this happened in the late 1990s on one of the Mars rovers. The solution was to restart the system. There are other solutions to the problem of priority inversion, such as temporarily boosting priorities.

A few other real-world examples In 2003, there was a blackout that affected around 55 million people in the eastern US and Canada. It was started by some sagging power lines, which wouldn't have been a problem except that there was a race condition in the monitoring software. That race condition prevent warnings from being seen until it was too late and the problem had spiraled out of control. In some places, it took weeks to restore power.

One of the biggest vulnerabilities in Linux was the *Dirty COW* vulnerability. It allowed full system takeover by running just a few dozen lines of code. The details are a little too much to get into here, but at the heart of it was a race condition.

One of the biggest security problems in recent years is the *Meltdown* vulnerability. It involves a design flaw in many Intel and ARM CPUs. The details, again, are a little more than we can get into here, but it involves another race condition.