Critical Sections & Race Conditions

This section exhibits how incorrect synchronization in a critical section can lead to race conditions and buggy code. The concepts of critical section and race condition are explained in depth. Also included is an executable example of a race condition.

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 program attempt to execute the same portion of code as explained in the following paragraphs.

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 based on the condition takes subsequent action. 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 takes action 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.

Consider the snippet below. We have two threads working on the same variable `randInt`. The modifier thread perpetually updates the value of `randInt` in a loop while the printer thread prints the value of `randInt` only if `randInt` is divisible by 5. If you let this program run, you'll notice some values get printed even though they aren't divisible by 5 demonstrating a thread unsafe version of **test-then-act**.

Example Thread Race

The below program 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.

Java

Example Thread Race

The below program 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.

Race Condition Example

Even though the if condition on **line 19** makes a check for a value which is divisible by 5 and

only then prints `randInt`. It is just after the if check and before the print statement i.e. in-between **lines 19** and **21**, that the value of `randInt` is modified by the modifier thread! This is what constitutes a race condition.

For the impatient, the fix is presented below where we guard the read and write of the `randInt` variable using the `RaceCondition` object as the monitor. Don't fret if the solution doesn't make sense for now, it would, once we cover various topics in the lessons ahead.

Java

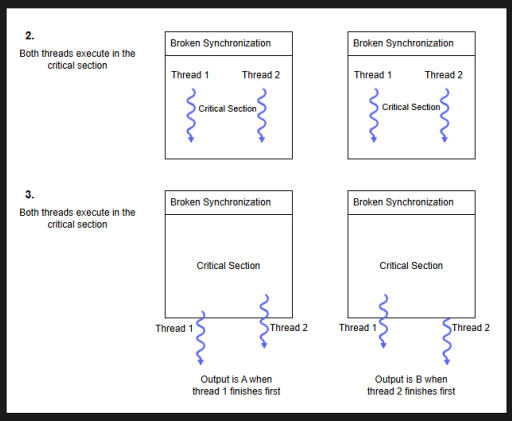
main.java — Saved

```
1  import java.util.*;
2
3  class Demonstration {
4
5      public static void main(String args[]) throws InterruptedException {
6          RaceCondition.runTest();
7      }
8  }
9
10 class RaceCondition {
11
12     int randInt;
```

Run

Below is a pictorial representation of what a race condition, in general, looks like.

Press + to interact



← Back lesson
Throughput vs Latency

Completed
Next →
Deadlocks, Liveness & Reentrant Locks