Critical Sections & Race Conditions

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

Critical Section & Race Conditions

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 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 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. The test 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 may not be divisible by 5, which demonstrates the existence of a race condition in the code.

If you run code widget below, scroll to the bottom of the output where you'll see a value not divisible by 5 for some of the runs.

S

```
$randInt = rand(1..100)

printerThread = Thread.new do

  while true

    if $randInt % 5 == 0
      puts("Divisible by five : #{$randInt}")
      if $randInt % 5 != 0
        puts("Not divisible by five : #{$randInt}")
      end
    end
  end
end

updaterThread = Thread.new do

  while true
```

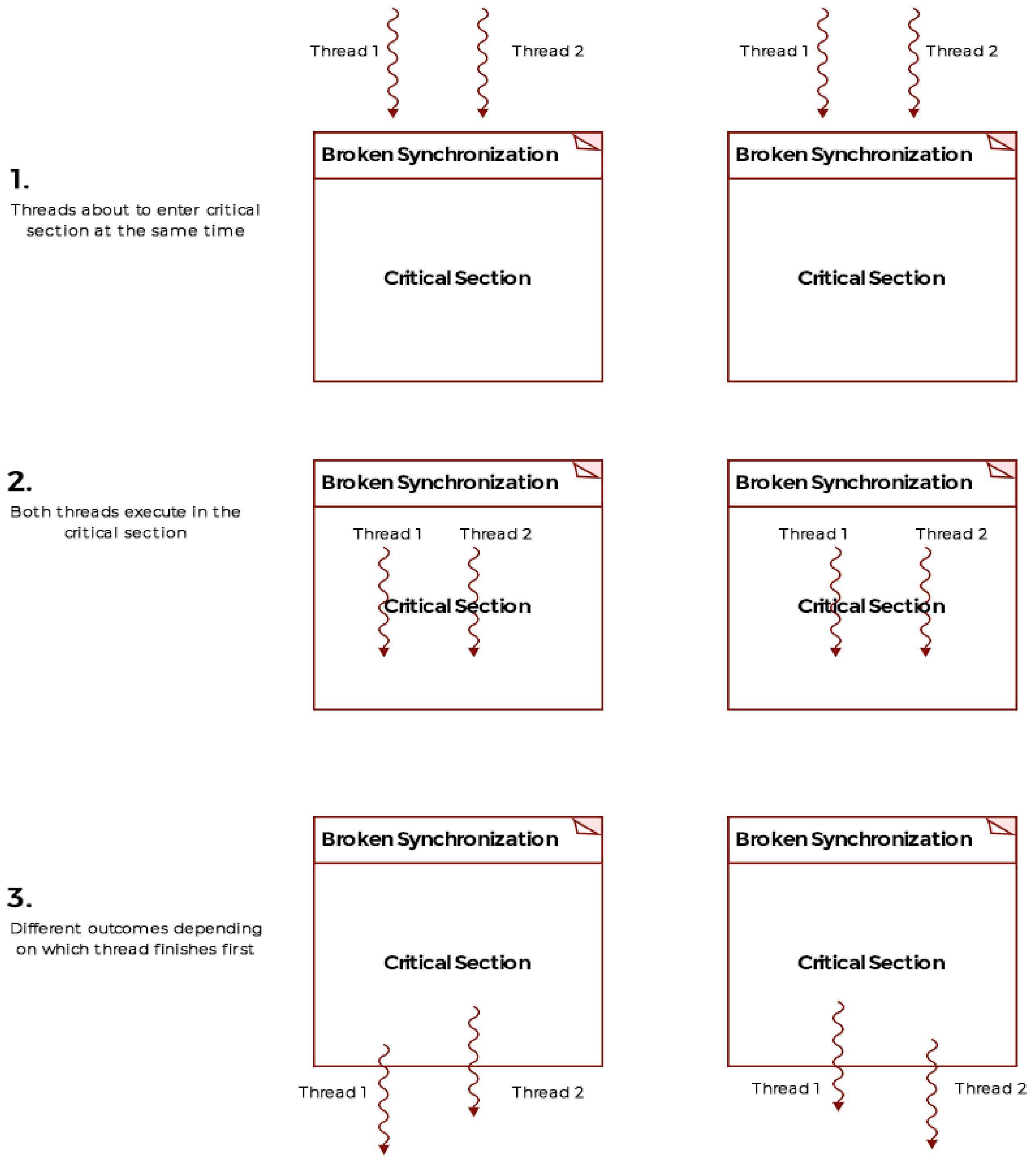
```
    $randInt = rand(1..100)
  end


end

# Let the simulation run for 20 seconds

sleep(20)
```

Even though the if condition on **line** 7 checks whether a value is divisible by 5 and only then prints `randInt`, but just ***after the if check and before the print statement***, i.e. in-between **lines** 7 and **10** 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 a `Mutex` object. Don't fret if the solution doesn't make sense for now. It will once we cover various topics in the lessons ahead. If you run the fixed version in the code widget below, there will be no output with a number that is divisible by 5.

```
$randInt = rand(1..100)
mutex = Mutex.new

printerThread = Thread.new do

  while true
    mutex.synchronize {
        if $randInt % 5 == 0
          puts("Divisible by five : #{$randInt}")
          if $randInt % 5 != 0
              puts("Not divisible by five : #{$randInt}")
          end
        end
    }
  end

end


updaterThread = Thread.new do

  while true
    mutex.synchronize {
        $randInt = rand(1..100)
    }
  end


end

# Let the simulation run for 20 seconds

sleep(20)
```

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



1.
Threads about to enter critical section at the same time
Thread 1
Thread 2
Broken Synchronization
Critical Section
2.
Both threads execute in the critical section
3.
Different outcomes depending on which thread finishes first

Output is A when thread 1 finishes first

Output is B when thread 2 finishes first