## Critical Section & Race Conditions

This lesson 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.

# 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.

## Example Thread Race

Consider the snippet below. We have two threads working on the same variable `rand_int`. The modifier thread perpetually updates the value of `rand_int` in a loop while the printer thread prints the value of `rand_int` only if `rand_int` 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**.

```
from threading import *
import random
import time

rand_int = 0


def updater():
    global rand_int
    while 1:
        rand_int = random.randint(1, 9)


def printer():
    global rand_int
    while 1:

        # test
        if rand_int % 5 == 0:
            if rand_int % 5 != 0:
                # and act
                print(rand_int)


if __name__ == "__main__":
    Thread(target=updater, daemon=True).start()
    Thread(target=printer, daemon=True).start()

    # Let the simulation run for 5 seconds
    time.sleep(5)
```

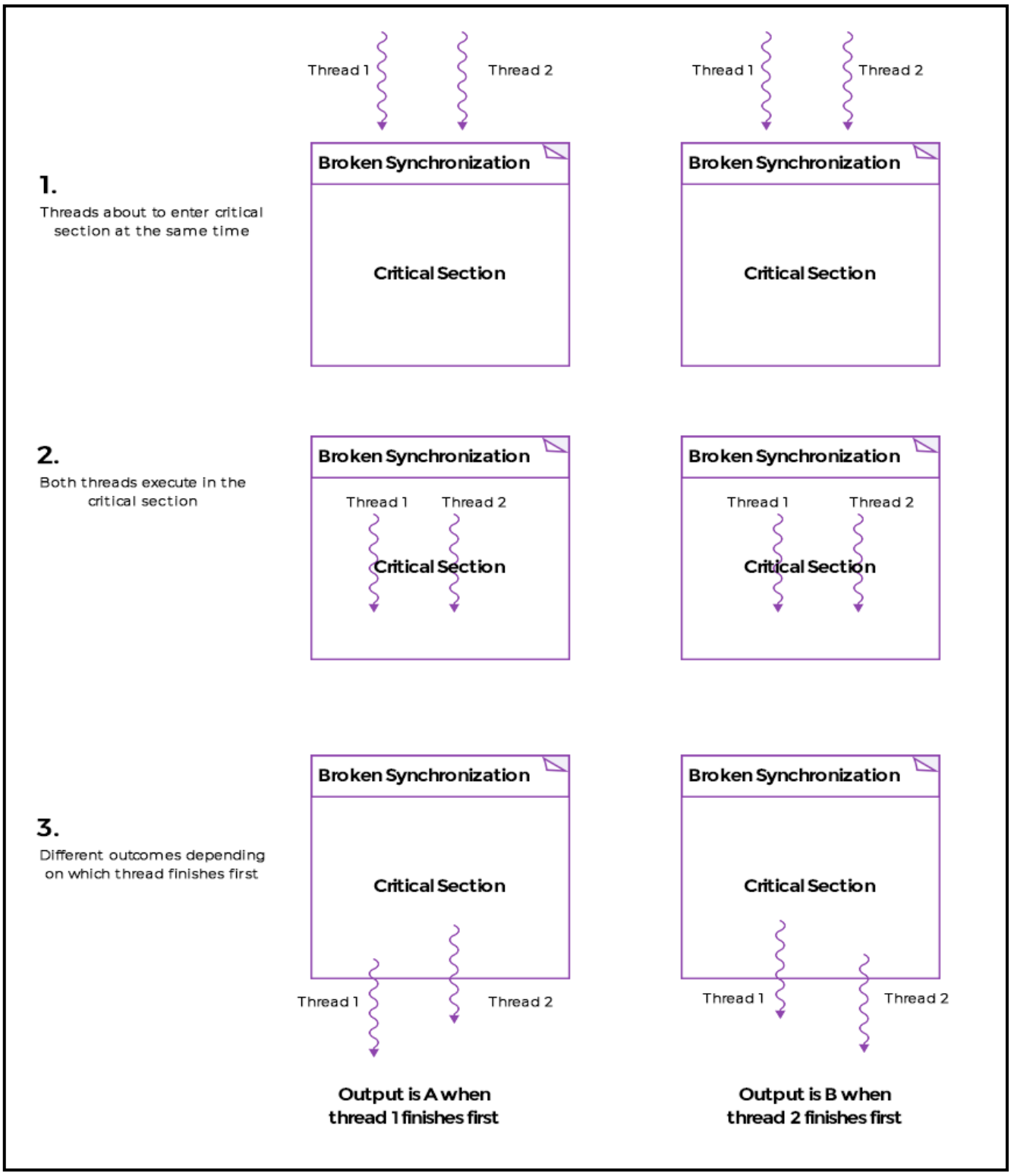
Even though the if condition on **line 19** makes a check for a value which is divisible by 5 and it only then prints `rand_int`, it is just ***after the if check and before the print statement***, i.e. in-between **lines 19** and **21**, that the value of `rand_int` is modified by the modifier thread! This is what constitutes a race condition.

For the impatient, the fix is presented below where we guard the read and write of the `rand_int` variable using a `Lock` object. Don't fret if the

solution doesn't make sense for now. It will once we cover various topics in the lessons ahead.

```
from threading import *
import random
import time

rand_int = 0
lock = Lock()

def updater():
    global rand_int
    global lock
    while 1:
        with lock:
          rand_int = random.randint(1, 9)


def printer():
    global rand_int
    global lock
    while 1:

        with lock:
          # test
          if rand_int % 5 == 0:
              if rand_int % 5 != 0:
                  # and act
                  print(rand_int)


if __name__ == "__main__":
    Thread(target=updater, daemon=True).start()
    Thread(target=printer, daemon=True).start()

    # Let the simulation run for 5 seconds
    time.sleep(5)
```

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




Back
Thread Safety
Next
Deadlock, Liveness & Reentrant Locks

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/critical-section-race-conditions__the-basics__python-concurrency-for-senior-engineering-interviews)