Why a simple flag variable cannot solve the problem of race condition?

Elaborating Problem and Solution

The crux of the problem of race condition is uncontrolled scheduling. When multiple threads execute the critical section, almost at the same time, then the threads update the shared variable or data structure and this results in surprising and undesirable results. This was described in the previous lecture video with the help of the following code example.
Code:

```
import threading

COUNT = 0

def calculate_count(arg):
  print("{}:begin".format(arg))
  global COUNT
  for _ in range(1000000):
      COUNT = COUNT + 1
  print("{}:ends".format(arg))

def main():
  print("main begin: COUNT = {}".format(COUNT))
  t1 = threading.Thread(target=calculate_count, args=("t1",))
  t2 = threading.Thread(target=calculate_count, args=("t2",))

  t1.start()
  t2.start()

  t1.join()
  t2.join()

  print("main done: COUNT = {}".format(COUNT))

if __name__ == "__main__":
  main()
```

Here, we can see there are two threads t1 and t2, which are simultaneously updating the shared variable COUNT. The solutions proposed in the lecture video were:

1. Atomic instruction: The three steps of, Get, Increment and Store, must be executed in one CPU cycle.
2. Mutual Exclusion: Only one thread should be allowed to execute the critical section. If one thread is executing the critical section, then other threads must wait, till the particular thread completes its execution.

The Natural Question Of Using Flag Variable

The second solution brings out the natural question that why can't we use a flag variable to ensure mutual exclusion among the threads. Let's use this approach to understand how this is insufficient.

So, we will use a simple flag variable to indicate whether the thread has acquired lock on the critical section. flag variable will have two values: 0 and 1 (initialized with 0 value). 0 indicates that the critical section is unlocked and the threads can enter in it, whereas 1 indicates that a particular thread has entered the critical section. The first thread that enters the critical section will test whether the value of flag is 1. Since, flag is set to 0, so, it will enter the critical section and set the flag to 1. When it is finished executing the critical section, then it will unlock the critical section by setting the flag to 0.

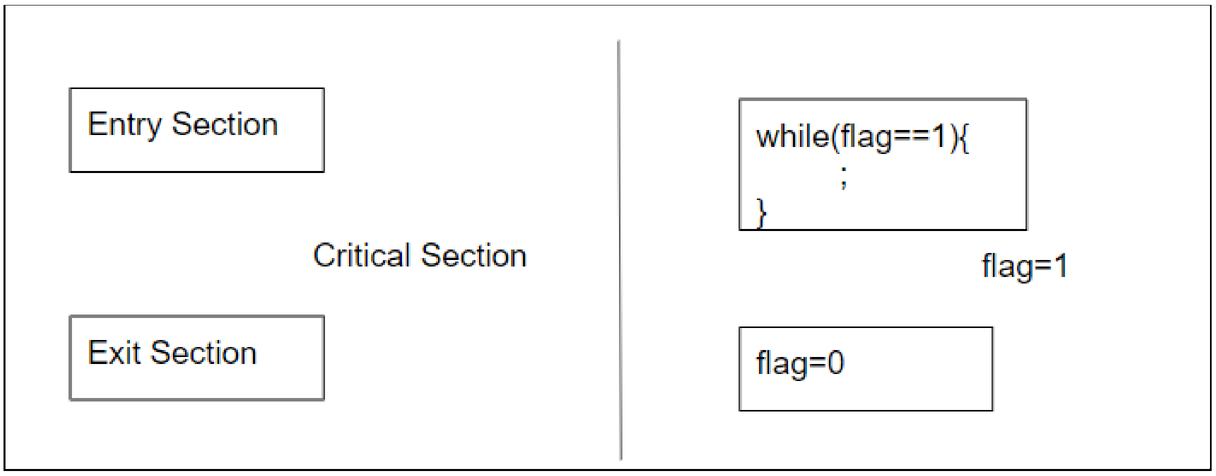
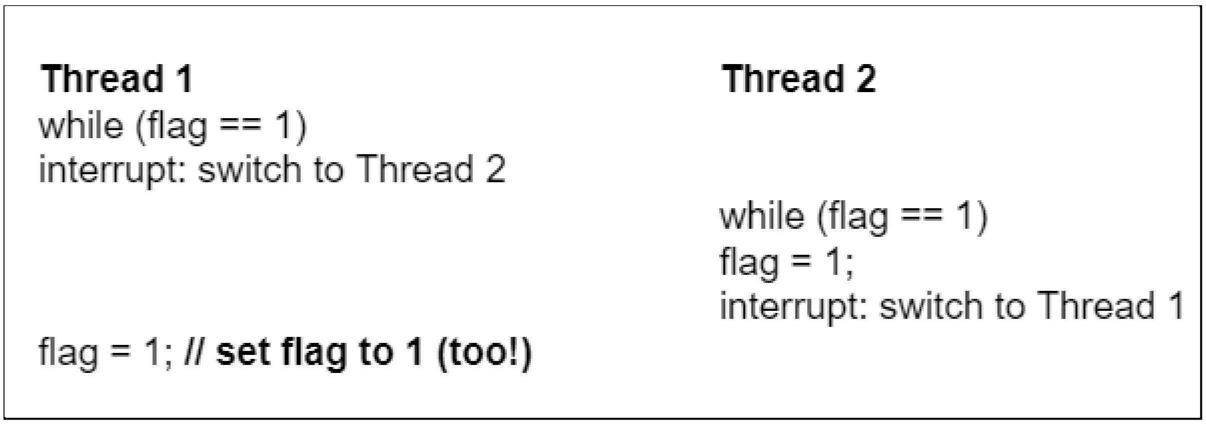


Figure 1

If another thread tries to enter the critical section, when the first thread is executing the critical section, then it will have to wait by executing the condition of while loop in the entry section to the critical section [as described in Figure 1]. Once the first thread completes the execution and sets flag to 0, then the waiting thread will fall out of the while loop, set flag to 1 and proceed to the critical section.

This is Failed Attempt!

Although this looks perfect, this solution is incorrect. The correctness issue can be understood simply by seeing the figure 2.



```
Thread 1                                   Thread 2
while (flag == 1)
interrupt: switch to Thread 2
                                           while (flag == 1)
                                           flag = 1;
                                           interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)
```

Figure 2

We have taken two threads: Thread 1 and 2. Initially, flag is set to 0 and thread 1 will start execution. When thread 1 completes the while condition execution and is about to set flag to 1, then it is preempted and control of execution moves to thread 2. Thread 2 will execute while loop’s condition and since, flag is still 0, thread 2 will set the flag to 1 and proceed to the critical section. Just then, thread 2 will be preempted and control of execution moves to thread 1. According to the flow of execution, thread 1 will again set the value of flag to 1 (**overwrite**) and enter the critical section. Hence, we have both the threads simultaneously executing the critical section and therefore, we have failed to provide mutual exclusion to the threads in this attempt!

Next Steps

The failed attempt may have dashed our hopes, but there is a software only solution to provide mutual exclusion. The solution was earlier provided by Dekker and it is called Dekker's algorithm. Dekker’s approach was later refined by Peterson and therefore, we will call it as Peterson’s Solution. Let us understand it in more detail in the next Aside note.