Week 2 - SYNC Assignment

Group 3

Table of contents

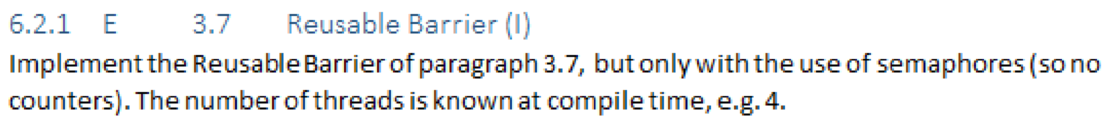
First Assignment
- **Problem description**
- **Solution**
- **Output**

Second Assignment
- **Problem description**
- **Solution**
- **Output**

Third assignment
- **Problem description**
- **Solution**
- **Output**

First Assignment

Problem description



6.2.1 E 3.7 Reusable Barrier (I)

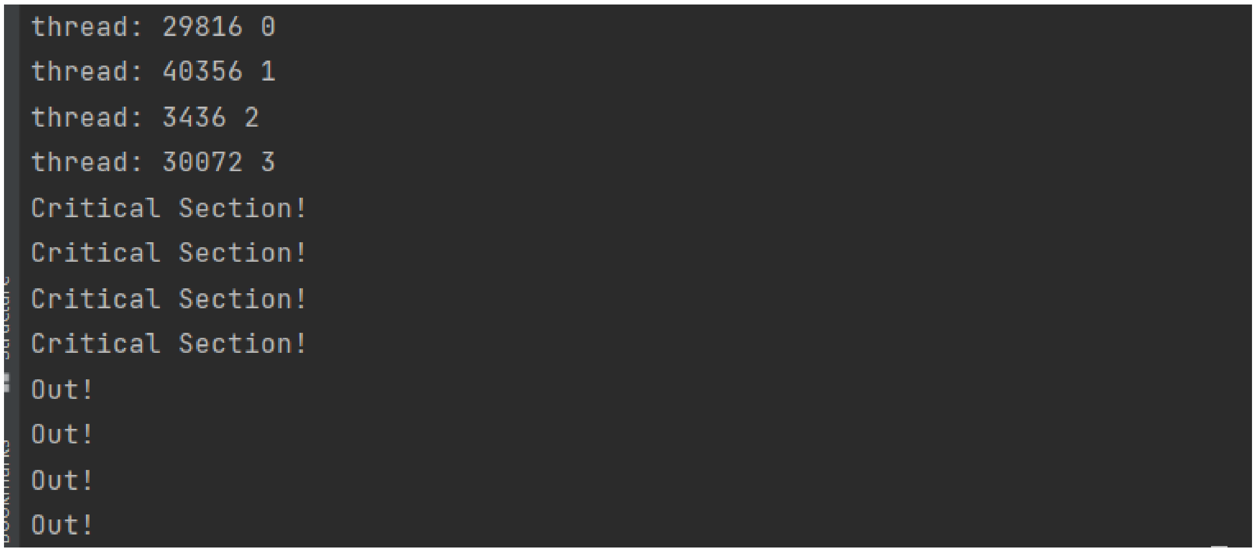
Implement the Reusable Barrier of paragraph 3.7, but only with the use of semaphores (so no counters). The number of threads is known at compile time, e.g. 4.

Solution

To solve this exercise, we have created 4 threads and 4 semaphores. In order to make sure that all threads get to the “Critical Section” at the same time, we make sure that within each thread, we signal all the semaphores and then a specific semaphore (e.g. semaphore 1) is waiting for the other threads to signal this specific semaphore.
Once all the threads are in “Critical Section”, in order to be out of it, we use the same principle.
The code for this assignment can be found below.

```
from Environment import *
semaphore1 = MySemaphore(0, "semaphore1")
semaphore2 = MySemaphore(0, "semaphore2")
semaphore3 = MySemaphore(0, "semaphore3")
semaphore4 = MySemaphore(0, "semaphore4")

def ThreadA():
  semaphore1.signal()
  semaphore2.signal()
  semaphore3.signal()
  semaphore4.signal()

  semaphore1.wait()
  semaphore1.wait()
  semaphore1.wait()
  semaphore1.wait()

  print("Critical Section!")

  semaphore1.signal()
  semaphore2.signal()
  semaphore3.signal()
  semaphore4.signal()

  semaphore1.wait()
  semaphore1.wait()
  semaphore1.wait()
  semaphore1.wait()
  print("Out!")


def ThreadB():
  semaphore1.signal()
  semaphore2.signal()
  semaphore3.signal()
  semaphore4.signal()

  semaphore2.wait()
  semaphore2.wait()
  semaphore2.wait()
  semaphore2.wait()

  print("Critical Section!")

  semaphore1.signal()
  semaphore2.signal()
  semaphore3.signal()
  semaphore4.signal()

  semaphore2.wait()
  semaphore2.wait()
  semaphore2.wait()
  semaphore2.wait()
  print("Out!")


def ThreadC():
  semaphore1.signal()
  semaphore2.signal()
  semaphore3.signal()
  semaphore4.signal()
  semaphore3.wait()
  semaphore3.wait()
  semaphore3.wait()
  semaphore3.wait()

  print("Critical Section!")

  semaphore1.signal()
  semaphore2.signal()
  semaphore3.signal()
  semaphore4.signal()

  semaphore3.wait()
  semaphore3.wait()
  semaphore3.wait()
  semaphore3.wait()
  print("Out!")


def ThreadD():
  semaphore1.signal()
  semaphore2.signal()
  semaphore3.signal()
  semaphore4.signal()

  semaphore4.wait()
  semaphore4.wait()
  semaphore4.wait()
  semaphore4.wait()

  print("Critical Section!")

  semaphore1.signal()
  semaphore2.signal()
  semaphore3.signal()
  semaphore4.signal()

  semaphore4.wait()
  semaphore4.wait()
  semaphore4.wait()
  semaphore4.wait()

  print("Out!")


def setup():
  subscribe_thread(ThreadA)
  subscribe_thread(ThreadB)
  subscribe_thread(ThreadC)
  subscribe_thread(ThreadD)
```

Output



```
thread:  29816 0
thread:  40356 1
thread:  3436 2
thread:  30072 3
Critical Section!
Critical Section!
Critical Section!
Critical Section!
Out!
Out!
Out!
Out!
```

A video is also included alongside this document.

Second Assignment

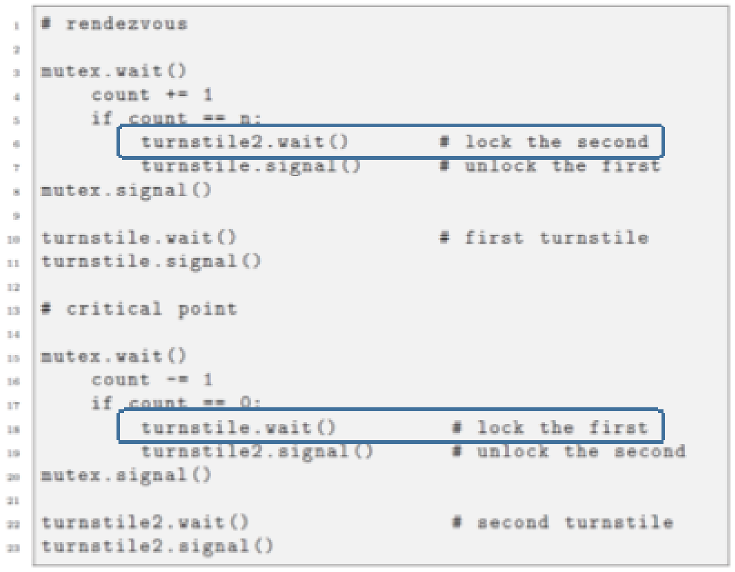
Problem description

6.2.2 F 3.7 Reusable Barrier (II)

Re-implement the solution of the Reusable Barrier of paragraph 3.7, but don't use `turnstile.wait()` for locking (aka closing) a turnstile (see the rectangles in the following picture).

Tip: do not start with the code as given in LBoS and move some statements around until it more or less seems to work, but start with an empty sheet and write a clean implementation.

Reusable barrier solution



```
# rendezvous

mutex.wait()
    count += 1
    if count == n:
        turnstile2.wait()         # lock the second
        turnstile.signal()        # unlock the first
mutex.signal()

turnstile.wait()                  # first turnstile
turnstile.signal()

# critical point

mutex.wait()
    count -= 1
    if count == 0:
        turnstile.wait()           # lock the first
        turnstile2.signal()        # unlock the second
mutex.signal()

turnstile2.wait()                  # second turnstile
turnstile2.signal()
```

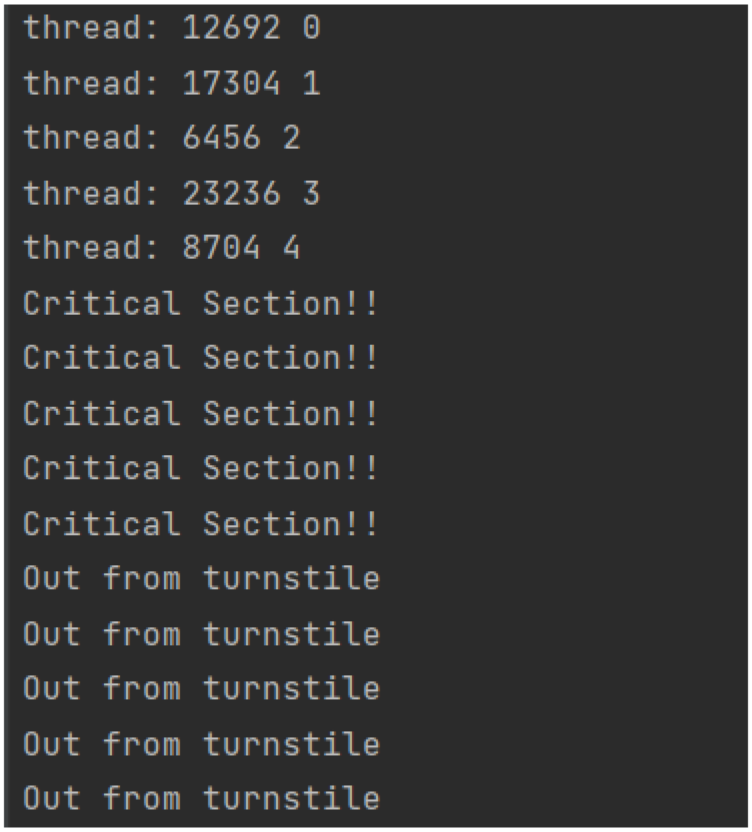
Solution

In this exercise, we used the solution of the book, but we made some changes to it in order to not use **turnStile.wait()** and **turnStile2.wait()**. When the counter is equal to the number of threads we signal all of them to enter the Critical section. The same logic applies when they have to exit the turnstile.

```
from Environment import *

N = 5
mutex = MyMutex(1)
turnStile1 = MySemaphore(0, "turnStile1")
```

```
turnStile2 = MySemaphore(1, "turnStile2")
count = MyInt(0, "count")
def ThreadN():
  mutex.wait()

  count.v += 1
  if count.v == N:
      turnStile1.signal(N)

  mutex.signal()

  turnStile1.wait()

  print("Critical Section!!")

  mutex.wait()
  count.v -= 1
  if count.v == 0:
      turnStile2.signal(N)

  mutex.signal()

  turnStile2.wait()

  print("Out from turnstile")


def setup():
  for i in range(N):
      subscribe_thread(ThreadN)
```

Output

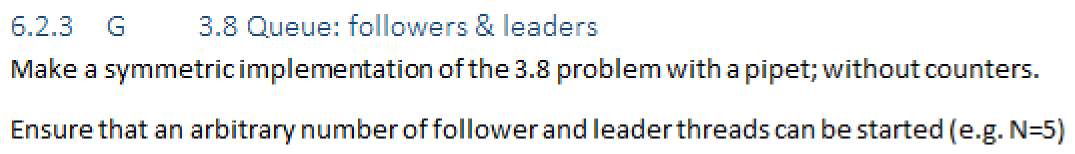


```
thread: 12692 0
thread: 17304 1
thread: 6456 2
thread: 23236 3
thread: 8704 4
Critical Section!!
Critical Section!!
Critical Section!!
Critical Section!!
Critical Section!!
Out from turnstile
Out from turnstile
Out from turnstile
Out from turnstile
Out from turnstile
```

A video is also included alongside this document.

Third assignment

Problem description


6.2.3 G 3.8 Queue: followers & leaders

Make a symmetric implementation of the 3.8 problem with a pipet; without counters.

Ensure that an arbitrary number of follower and leader threads can be started (e.g. N=5)

Solution

In this exercise, we managed to do another implementation for “followers & leaders”, by using a pipet. Firstly, we initialized 2 mutexes and 2 semaphores, the mutexes representing the pipet for the followers and leaders, and the semaphores representing the rendezvous. We also created 2 new threads one for leaders and one for followers. When we execute one of them for instance “leader”. It will first signal to one follower and then it waits for the follower to signal. If both of them signal, then they will access the critical section together.

```
from Environment import *
N = 4
leaderPipet = MyMutex("leaderPipet")
followerPipet = MyMutex("followerPipet")
leaderRendezvous = MySemaphore(0, "leaderRendezvous")
followerRendezvous = MySemaphore(0, "followerRendezvous")

def leadersThread():
  while True:
    leaderPipet.wait()

    followerRendezvous.signal()
    leaderRendezvous.wait()
    print("Dance")

    leaderPipet.signal()

def followersThread():
  while True:
    followerPipet.wait()
    leaderRendezvous.signal()
    followerRendezvous.wait()
    print("Dance")

    followerPipet.signal()
```

Output

A video is also included alongside this document.