## Week 2

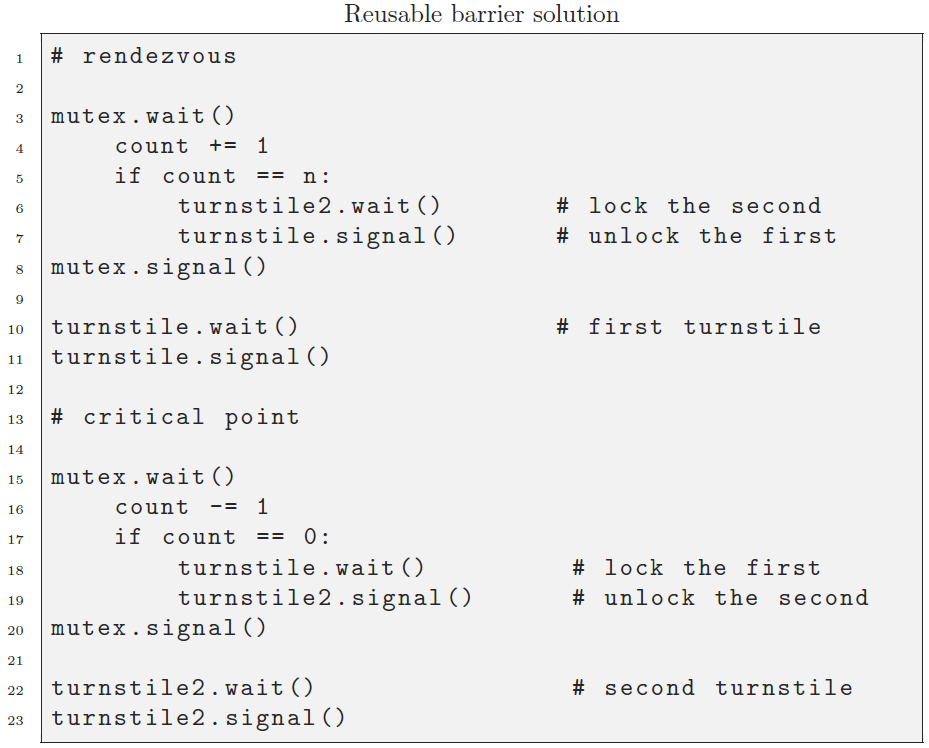
### E 3.7 Reusable Barrier (I)

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.

### 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.



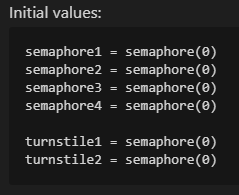
## Answer

For these two assignments, we have designed four different solutions. These solutions are first an asymmetric solution. The second solution uses variables and semaphores but is symmetric. The third uses only semaphores. Our fourth and final solution also only uses semaphores. We did receive a few hints from Joris to come up with it.

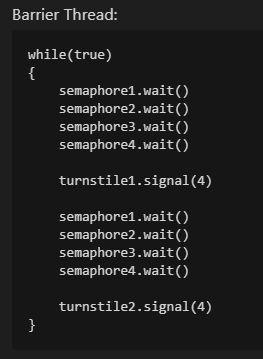
**Asymmetric solution:**

Our first solution was done using asymmetry and involved n+1 threads. The extra threads are in charge of scheduling when the other n threads are allowed to proceed. The code for this solution can be found in the file: assignment\_EF\_assymetric.md

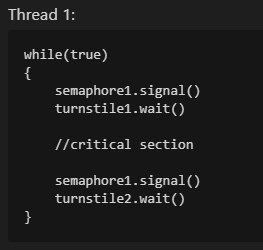
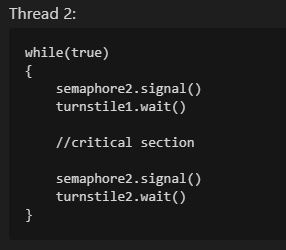
The first solution used two different turnstiles and four semaphores:

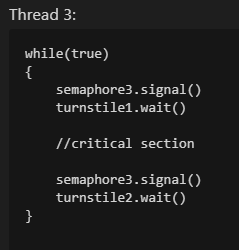
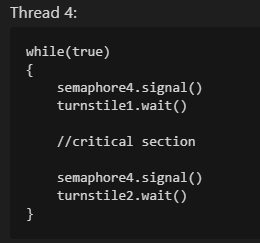


What this solution does is have a single Barrier thread that only makes all the other threads wait. This is how that thread looks:



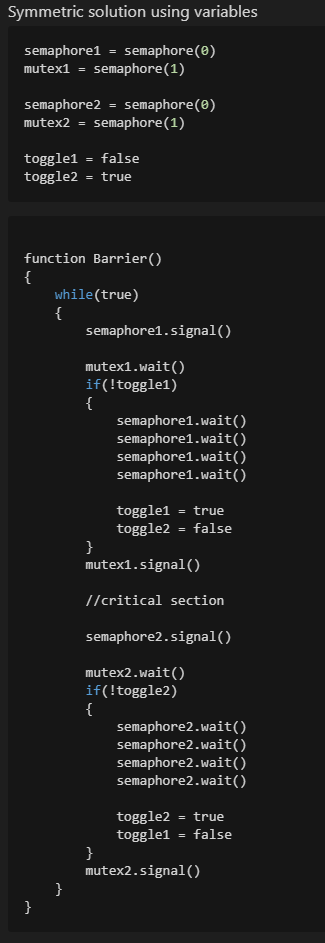
Then the other thread signals the barriers threads semaphores. Then after they single they will wait for the turnstile. When all threads have arrived the barrier thread preloads the turnstiles for the others:

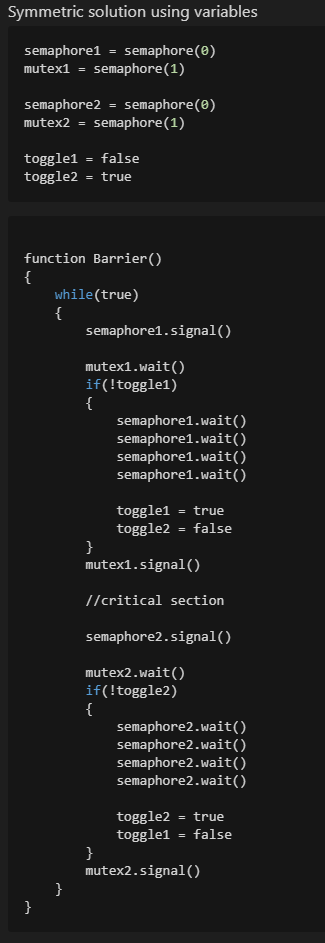




**Variable solution**

Our second solution has several implementations. One using semaphores and variables, and one using just variables. The underlying mechanism for both is the same, in that they involve calling wait() on a specific semaphore numerous amount of times until the right amount of signals have been given by other waiting threads. These two implementations can be found in the file: assignment\_EF\_symmetric.md

The variable solution requires two basic semaphores and two mutexes. The variables we use are two toggles. They are used to skip the wait steps in this solution. {

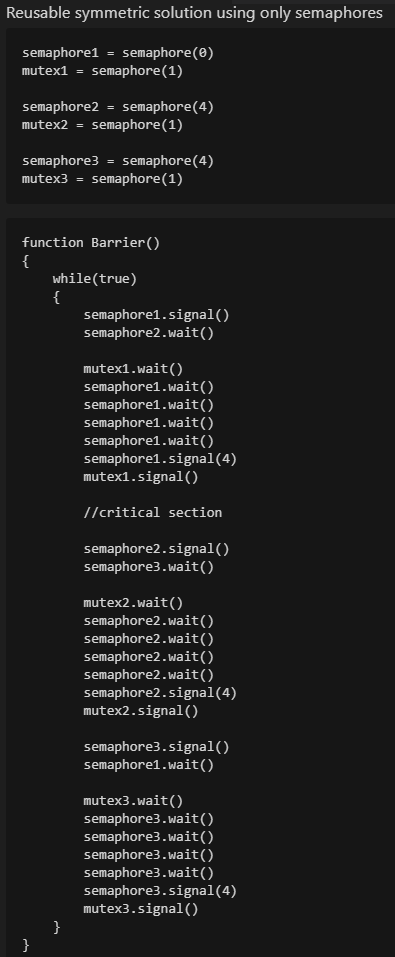
This solution works by having the first thread wait for the others to signal the semaphores the first thread is waiting for. The toggle is used so that the other threads do not have to wait.

Then all the other threads can pass by. We will then also toggle the two variables.

To make the barrier reusable we duplicated the barrier but used the second toggle instead.

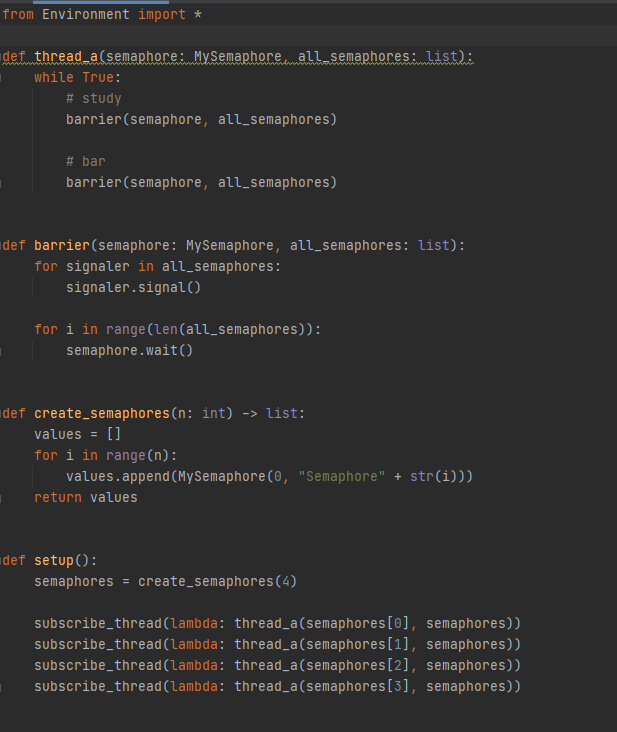
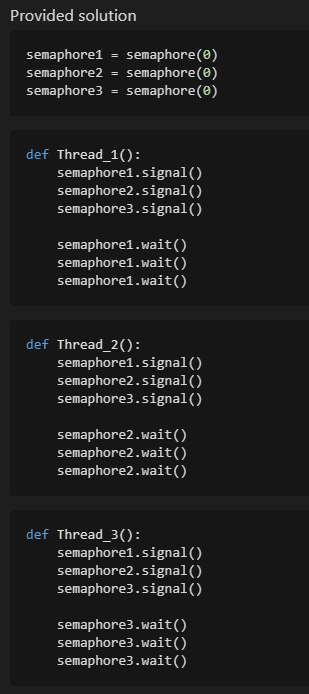
**Semaphore only solution**

This version uses a third semaphore instead of two toggle variables. It adds a third barrier that allows us to use to reset the first barrier.



**Basic semaphore solution**

During the Friday session for Synchronization with a few hints from Joris. We were able to figure out the basic version. To make it as symmetric as possible we decided that with the python implementation we would specify the semaphores that were used. This allows us to be able to have an unspecified amount of semaphores. This is the code that we wrote for it:



When you move all the signals and waits outside of loops and functions you will get that each thread executes steps like this.

### 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)

## Answer

For this, we created a solution in which the pipet acts as both the mutex as well as the queue. Inside of this pipet, we use a rendezvous to align the two threads with one another. The code for this can be found in the file: assignment\_G.md

