Module D3: Synchronization

Contents

[1 Introduction 1](#_Toc88556830)

[2 Learning outcomes 1](#_Toc88556831)

[3 Resources 2](#_Toc88556832)

[4 Teaching and grading 2](#_Toc88556833)

[5 Schedule 2](#_Toc88556834)

[6 Assignments 2](#_Toc88556835)

[6.1 Week 1 2](#_Toc88556836)

[6.2 Week 2 4](#_Toc88556837)

[6.3 Week 3 5](#_Toc88556838)

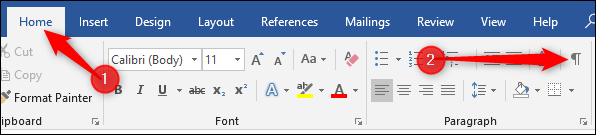
[6.4 Week 4 6](#_Toc88556839)

[6.5 Week 5 6](#_Toc88556840)

# Introduction

One cannot board a plane before buying a ticket. Similarly, one cannot check e-mails before validating the appropriate credentials (for example, user name and password). In the same way, an ATM will not provide any service until it receives a correct PIN.

Daily life is full of synchronization, computer systems as well. Troubles are very subtle: solutions may seem to work in most of the cases, but we are only interested when they work properly in all cases. Remember Murphy's Law: “Anything that can go wrong will go wrong“.

Some exercises have hints. You can view them via Ctrl-Shift-8, or: 

# Learning outcomes

In addition to the general Academic Preparations learning outcomes, the student is able to:

* understand, recognize and solve the problems around synchronization between threads like mutual exclusion and deadlock
* apply the common primitives like mutexes, semaphores, conditional variables
* apply common synchronization design patterns
* build a visual metaphor of a problem

# Resources

* <https://greenteapress.com/wp/semaphores/>
* <https://git.fhict.nl/I878848/SyncSimulator>

# Teaching and grading

The slides (SYNC.pdf) contain global introductions to weekly topics. For the actual study, the book Little Book of Semaphores is used.

Be aware: the book already contains solutions of the problems. First try them yourself, and please note that the solutions are not always optimal. In this course we sometimes modify the problems, or we use other primitives to solve them (like condition variables).

Your grade is based on:

* an on-campus assignment at the end
* the home assignments

Delivery of the home assignments:

* all assignments have to be delivered in Canvas, before the deadline
* to be delivered:

1. well formatted source files
2. reasoning about the correctness
3. screen shot of the output and/or video with commentary
4. note: when there are e.g. two parties involved with an identical structure (like: producers and consumers): do not duplicate their code, but use parametrization

Remember: the lecturer wants to measure your intelligence and he has no telepathic capabilities, so proofs of your intelligence have to be given by explanations in additional documents, screen shots and videos.

All assignments (home + on-campus) have to be sufficient.

# Schedule

|  |  |  |
| --- | --- | --- |
| week | preparation | assignments |
| 1 | ch 1 - 3.5 of Little Book of Semaphores | see paragraphs below |
| 2 | par. 3.6 - 3.8 |  |
| 3 | par. 4.1 - 4.4, 5.5 - 5.8 |  |
| 4 | par. 5.1 - 5.4  <https://hpc-tutorials.llnl.gov/posix/condition_variables>  <https://hpc-tutorials.llnl.gov/posix/waiting_and_signaling> |  |
| 5 | par. 6.1 - 6.4 |  |
| 6 |  |  |

# Assignments

## Week 1

### A Critical Section

There are two threads, with the code listed below. The programmer did his best to achieve mutual exclusion, fairness, and to avoid deadlock. However, m*utual exclusion* is not guaranteed.

Answer the following questions:

1. how can both threads enter the Critical Section at the same time?
2. can *deadlock* occur? (why/why not?)
3. is this implementation *fair* (i.e. is it *starvation*-free)?

For all three questions: give a precise description how it happens. Make a table where you write on each line the executed statement (like A1 or B7), together with the actual value of the variables (flag[0], flag[1], lock[0], lock[1]) after execution of that statement.

bool flag[2] = { false, false };   
bool lock[2] = { false, false };

***thread A: thread B:***

while (true) while (true)

{ {

1. flag[0] = true; flag[1] = true;
2. lock[1] = false; lock[0] = false;
3. if (flag[1] == true) if (flag[0] == true)

{ {

1. lock[1] = true; lock[0] = true;
2. flag[0] = false; flag[1] = false;

} }

1. while (lock[1] || flag[1]) while (lock[0] || flag[0])

{ {

1. flag[0] = false; flag[1] = false;
2. flag[0] = true; flag[1] = true;

} }

1. CriticalSection(); CriticalSection();
2. flag[0] = false; flag[1] = false;
3. lock[0] = false; lock[1] = false;

} }

### B Interleaving

Given the following statements:

x = 0  
def myThread():  
 global x  
 for i in range(100):  
 x += 1

myThread is started two times. They both execute the for-loop such that x will be incremented.

The operation x += 1 is not atomic; in assembler code it could be something like:

for one thread:

load R1, @x  
 inc R1  
 store R1, @x

for the other thread:

load S1, @x  
 inc S1  
 store S1, @x

(R1 and S1 are registers of the CPU)

Because those instructions are not secured with semaphores, strange situations can happen with the contents of x. If everything runs sequentially in a proper way, then we expect that x has afterwards a value of 200. A larger value than 200 is not expected.

The assignment:

* It appears that there is a scenario that x is 2 at the end of the process. Design this scenario (watch out: this requires a creative brain!!!!).
* If you cannot find such a scenario, what's the lowest value that you have discovered? (200?, 101?, 100?, 1?, …?)
* Describe how the threads are interleaving their statements to reach that value of x.

### C Synchronization

Create and run 4 threads A, B, C and D.

They print the numbers 1 until 8 on one terminal. Thread A prints the number 1 and 5, thread B prints 2 and 6, thread C prints 3 and 7, thread D prints 4 and 8.

Requirements:

* the semaphores may be created before the threads are started
* the numbers are printed in the "right order"
* you may only use semaphores for synchronization (so no busy-wait loops, no shared memory)
* it should not make any difference in which order the threads are started

### D Deadlock

Create three threads and three semaphores. Write synchronization code with the risk of deadlock, but where they also may run for hours without problems.

Implement in the simulator and demo the deadlock and the smooth operation.

|  |
| --- |
| Hint:  Look at the slide “Deadlock (II)”. How can these threads run smoothly, and how can deadlock finally occur? The code for this assignment highly resembles the code of the slides. |

## Week 2

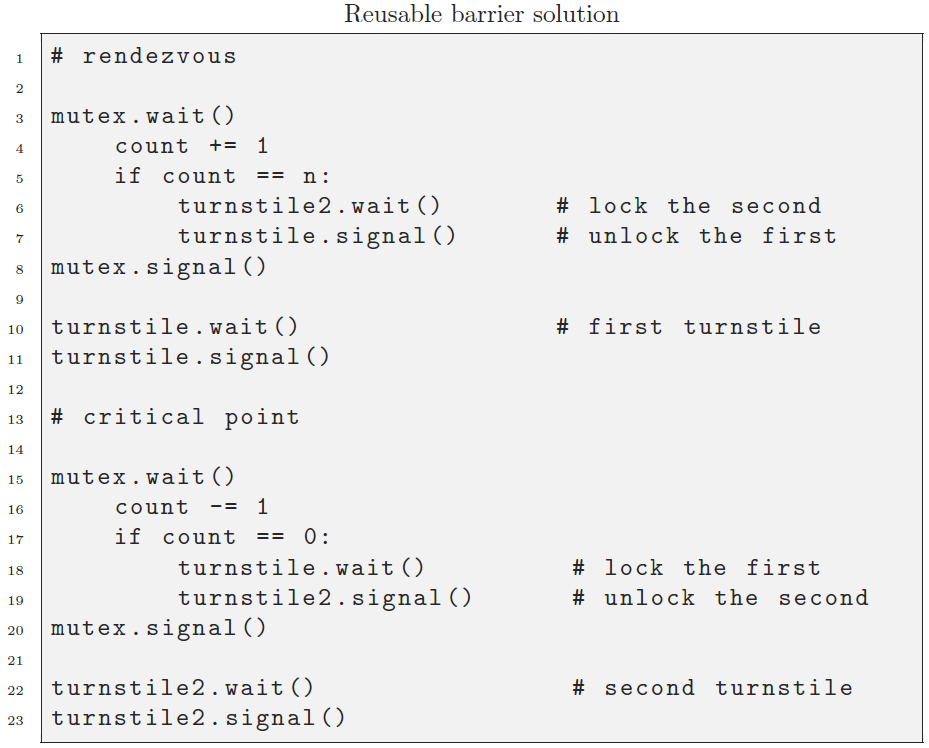
### E 3.7 Reusable Barrier (I)

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.



### 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)

## Week 3

### H 4.4.4 dining philosophers (I)

Implement the solution as suggested in 4.4.4 such that there is no circular-waiting (take the standard 4.4 solution as your starting point).

### I 5.5 Santa Clause

Implement with slightly different requirement: with helping *at least* 3 elves, and reindeer don't have priority. Furthermore: the elf's getHelp() can only be executed together with Santa's helpElves() (so: an explicit synchronization has to be added which is not listed in the book)

Ensure that an arbitrary number of elf threads can be started (e.g. N=7)

Tips:

* check first what goes wrong in the LBoS-solution when the reindeer/elf priority has been swapped
* 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

### J 5.6 H2O

Implement without counters (but semaphores, mutexes, pipets, queues, barriers are allowed).

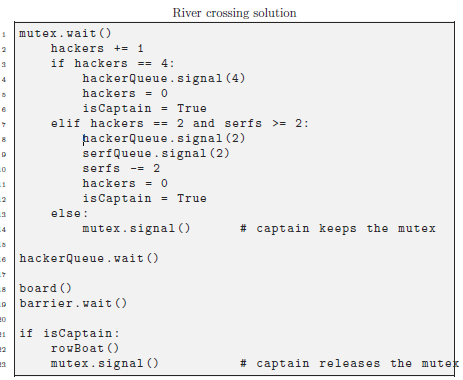
Ensure that an arbitrary number of H and O threads can be started (e.g. N=7).

### K 5.7 river crossing

Implement with a symmetric mutex usage (but counters, semaphores, mutexes, pipets, queues, barriers are allowed).

Ensure that an arbitrary number of hacker and serf threads can be started (e.g. N=7).

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.



## Week 4

### L 4.4 dining philosophers (II)

Implement a deadlock free solution with condition variables, by avoiding the *hold-and-wait* deadlock condition.

|  |
| --- |
| Hint:  use Booleans to indicate if the forks are in use and condition variables to notify changes in the status of the forks. There is no need for separate mutexes for each fork. |

### M 5.1 dining savages

Implement with condition variables with the following requirements:

* real storage for servings (use MyBag)
* carnivore + vegetarian savages
* carnivore + vegetarian cooks

So the pot contains a variety of servings, and there are a variety of savages around the pot. And sometimes it might occur that the pot does contain servings, but not suited for the available savages.

Ensure that an arbitrary number of carnivore-savages and vegetarian-savages can be started.

### N 4.2 readers-writers

Implement with condition variables, and make it configurable who has priorities.

Ensure that an arbitrary number of reader and writer threads can be started (e.g. N=7).

|  |
| --- |
| Hint:  mutex = MyMutex("mutex") cv\_reader = MyConditionVariable(mutex,"cv\_reader") cv\_writer = MyConditionVariable(mutex,"cv\_writer") nrof\_reader\_busy = MyInt(0,"reader\_busy")  nrof\_writer\_busy = MyInt(0,"writer\_busy") nrof\_reader\_wait = MyInt(0,"reader\_wait") nrof\_writer\_wait = MyInt(0,"writer\_wait") writer\_prio = MyBool(False,"writer\_prio") |

## Week 5

### O 6.3 baboon crossing

Implement with states and semaphores, and without a light switch.

Ensure that an arbitrary number of north-side-baboons and south-side-baboons can be started. As they have identical behavior, implement only one thread-function (e.g. threadBaboon(me,other)) where me and other contain semaphores and counters (etc.) for its own side and the other side.

|  |
| --- |
| Hint:  state = MyString("EMPTY", "state") # other states: “NORTH”, “SOUTH” mutex = MyMutex("mutex") capacity = MySemaphore(5, "capacity") northCount = MyInt(0, "nCount") northCandidates = MyInt(0, "nCand") # northSem acts like a ‘queue’, where either you open it for yourself, or it is opened by the south baboon (when he leaves) northSem = MySemaphore(0, "nSem")) southCount = MyInt(0, "sCount") southCandidates = MyInt(0, "sCand") southSem = MySemaphore(0, "sSem")) |

### P 6.4 modus hall with errors

For simplicity reasons, the first arriving Prude already triggers the transition to Prudes.

1. given Dut64\_ModusHall\_CondVar\_Error.py:  
   investigate what goes wrong, and when it goes wrong
2. modify Dut64\_ModusHall\_CondVar\_Error.py: when you are in transition (e.g. TRANS\_TO\_PRUDE) and the last person leaves (e.g. a Heathen), then directly go to the other’s active state (e.g. PRUDES\_RULE)   
   investigate what goes wrong, and when it goes wrong

### Q 6.4 correct modus hall

Based on your investigations of the previous exercise, copy Dut64\_ModusHall\_CondVar\_Error.py into Dut64\_ModusHall\_CondVar.py and write a correct implementation of the Modus Hall problem with condition variables and states.

|  |
| --- |
| Hint:  do the transition in two phases (first phase: when the first Prude arrives while Heathens are walking; second phase: when the last Heathen has left the path) such that you can ensure that a waiting Prude can really take the path (ie. enter the Critical Section). |

Note: the LBoS-solution of 6.4.2. contains a bug…