# Exam SYNCHRONIZATION

name:

Date: Tue 21 Jun 2022

Time: 16:00-17:30

Location: R10

Teacher: J. Geurts

Resources: none

Grading: 100 points (+ 20 points bonus)

|  |
| --- |
| General remark: if you think that the questions are not precise enough to solve it, then you can add your own assumptions and continue your work.  Please clearly write down your assumptions and the reason for adding them. |

1. Terminology (20p)

Describe the following concepts:

* race condition
* mutual exclusion
* starvation
* busy waiting

For each: give a computer world example (with threads etc.) and a real world example.   
We expect an answer of a few lines.

# No-starve mutex (20p)

Consider the following code.

|  |
| --- |
| N = 5  sem = MySemaphore(1,"sem")  def myThread():  while True:  sem.wait()  # critical section  sem.signal()  # other non-critical stuff  setup():  for in in range(N):  subscribe\_thread(myThread) |

Note: the semaphore is a *weak* semaphore (i.e. if there are threads waiting on a semaphore when a thread executes signal(), then *one* of the waiting threads will be woken, but there is no guarantee *which* of the waiting threads it will be)

Assignments:

1. describe a scenario for the code above that there is starvation possible for one thread (or more)
2. based on weak semaphores, how can you make it starvation-free?  
   you can give a code implementation, or a clear description   
   here you may assume that there are no more than N threads in the system

# Class room (30p)

Given the following scenario:   
Each day, a teacher gives lectures to a group of students.  
They arrive in the morning. The teacher unlocks the door of the class when there are 3 (or more) students present.   
Then he starts the lecture. (note: the students are not from Fontys: it might happen that they come in late when the lecture already has started, and it might happen that they quit the lecture and the class room before the lecture has finished…)  
After the lecture, he locks the door, but only after *all* students have left the class room.   
And they all go home.

You may assume that *all* students come to the class each day.

The *unsynchronized* code of the teacher and the students is:

|  |  |
| --- | --- |
| NROF\_STUDENTS = 30  def thread\_teacher():  while True:  go\_to\_school()  unlock\_class\_room()  give\_lecture()  lock\_class\_room()  go\_home() | def thread\_student():  while True:  go\_to\_school()  enter\_class\_room()  attend\_lecture()  leave\_class\_room()  go\_home() |
| setup():  subscribe\_thread(thread\_teacher)  for in in range(NROF\_STUDENTS):  subscribe\_thread(thread\_student) | |

Assignment:

* Synchronize the threads (you may use semaphores, mutexes, barriers, condition variables, integers, booleans)  
  Obey the common rules as discussed in the SYNC lectures
* Note: you don’t have to give the implementation of the functions like go\_to\_school() etc, but only the synchronization *between* those functions.

# Synchronize (10p)

There are two threads **T0** en **T1** and they have the following shared variables (with their initial values):

turn = 0

flag0 = False

flag1 = False

*thread T0: thread T1:*

while True: while True:

*A* if flag1: *R* if flag0:

*B* turn = 1 *T* turn = 0

else: else:

*C* turn = 0; *U* turn = 1

*D* flag0 = True *V* flag1 = True

*E* while flag1 **or** turn==1: *W* while flag0 **or** turn==0:

*F* pass *X* pass

*G* critical\_section() *Y*  critical\_section()

*H* flag0 = False *Z* flag1 = False

Deadlock can happen.

Assignment:

* give a precise scenario how it can happen

Please note: this example is *not* exactly identical to the slide of the lectures…

For writing the scenario, please use the table on the last page.

Put the executed statement letter in the first column (A till H if thread T0 executes a statement, **or** R till Z if thread T1 executes a statement)

And put in the other columns the actal values of the variabeles turn, flag0, and flag1 **after** the execution of that statement.

# Print (20p)

There are two threads **P0** en **P1** and they have the following shared variables (with their initial values):

x = 10

*thread P0: thread P1:*

while True: while True:

*J* x = x - 1 *N* x = x - 1

*K* x = x + 1 *O* x = x + 1

*L* if not x == 10: *P* if not x == 10:

*M* print(“x is ”, x) *Q* print(“x is ”, x)

The threads run in parallel. Please remember that a python statement like x=x-1 and x=x+1 are executed in three machine code instructions, like:

x += 1: LOAD $R1, @x x -= 1: LOAD $R1, @x  
 INCREMENT $R1, 1 DECREMENT $R1, 1  
 STORE $R1, @x STORE $R1, @x

Assignment:

1. (10pt) give the scenario such that “x is 10” will be printed
2. (10pt) give the scenario such that “x is 8” will be printed
3. (10pt BONUS) give the scenario such that “x is 12” will be printed

Please note: these are three different assignments. They all start from the initial situation.

For writing the scenario’s, please use the table on the last page.

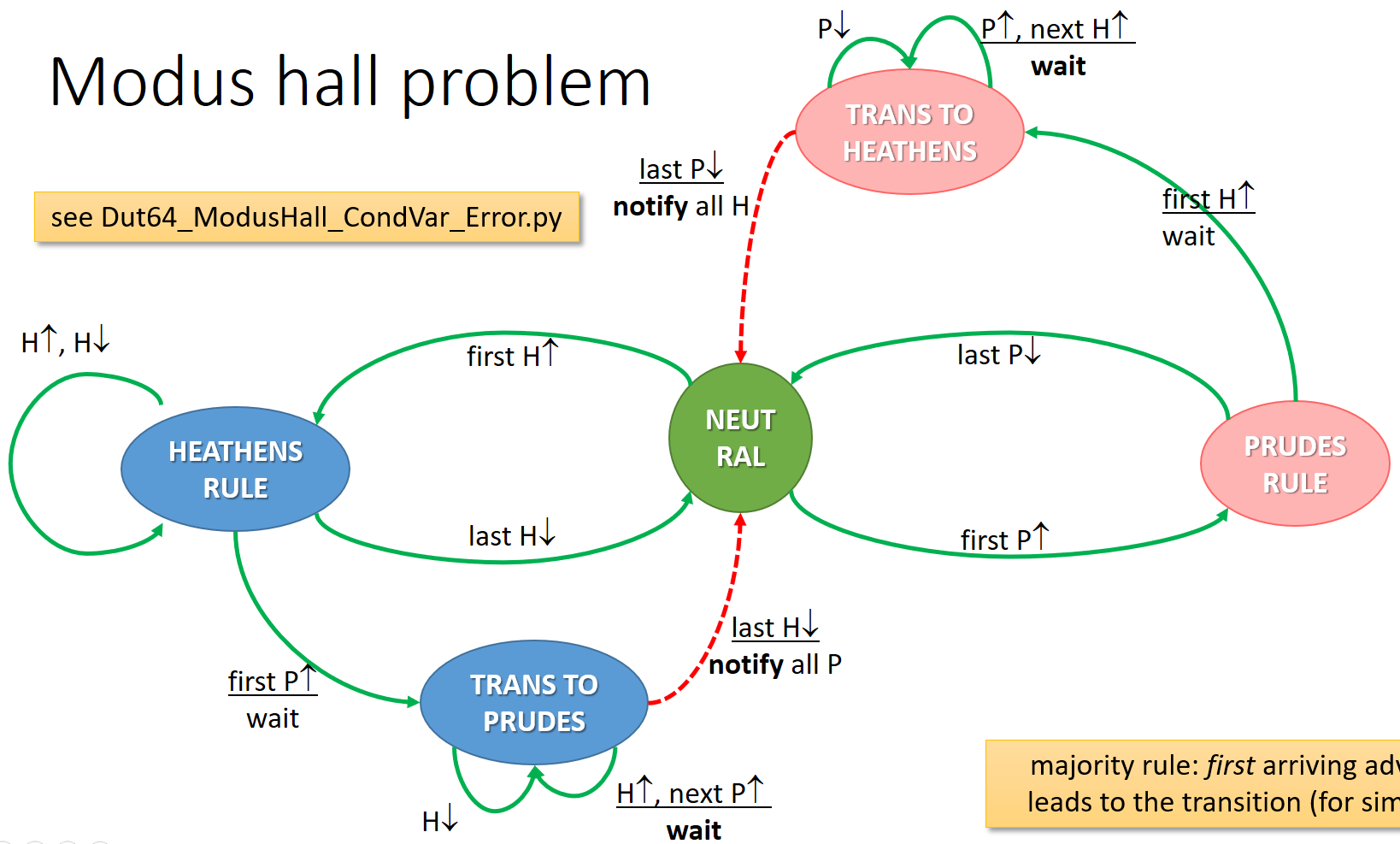
Put the executed statement letter in the first column (J till M if thread P0 executes a statement, **or** N till Q if thread P1 executes a statement)

And if only a part of a statement is executed (i.e. one machine instruction), write it like: A.LOAD or B.INC or N.STORE.

# Bonus: Modus Hall (10 pt)

Given is the following state-transition-table and implementation of the Modus Hall Problem (it’s identical to the one of the SYNC lectures):

|  |
| --- |
| N = 4  def threadPerson(me, other):  while True:  mutex.wait()  if state.v == other.state\_walk:  state.v = other.state\_trans  while not (state.v=="NEUTRAL" or state.v==me.state\_walk):  me.cv.wait()  state.v = me.state\_walk    me.count.v += 1  mutex.signal()  # CS    mutex.wait()  me.count.v -= 1  if me.count.v == 0:  state.v = "NEUTRAL"  other.cv.notify\_all()  mutex.signal()  class Person(object):  def \_\_init\_\_(self, count, cv, state\_walk, state\_trans):  self.count = count  self.cv = cv  self.state\_walk = state\_walk  self.state\_trans = state\_trans  state = MyString("NEUTRAL)  mutex = MyMutex()  heathen = Person (MyInt(0, "heathenCount"),  MyConditionVariable(mutex, "heathenCV"),  "HEATHENS\_RULE", "TRANS\_TO\_PRUDES")  prude = Person (MyInt(0, "prudeCount"),  MyConditionVariable(mutex, "prudeCV"),  "PRUDES\_RULE", "TRANS\_TO\_HEATHENS")  def setup():  for i in range(N):  subscribe\_thread(lambda: threadPerson(heathen, prude))  for i in range(N):  subscribe\_thread(lambda: threadPerson(prude, heathen)) |



Describe what goes wrong, and how it happens (e.g. by giving a sequence of events).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **statement** | **turn** | **flag0** | **flag1** |  | **statement** | **x** |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |