Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Mohammadreza Hazhirpasand

Solution Concurrent Programming - Exam

Given Name: ________________

Family Name: ________________

Matriculation Number: ________________

Please read this page carefully and fill in your name right away!

Examination date:	Wednesday, 18 December 2019
Version:	1.1
Time:	60 minutes
Total points:	60 (one point requires about one minute of work)
Number of exercises:	5
Allowed materials:	a single ink-based pen (no pencil)

Please remember:

1. Some exercises are split over more than one page. Please quickly skim through all exercises first, before you proceed with the exam.
2. If you don't know something don't waste any time, go ahead and try to solve those questions at the end.
3. You are expected to answer each question concisely (brief and precise).
4. If you need more blank space for your answers, please use the back of your pages, and clearly state to which question your answer relates.
5. Please pay attention that your handwriting is easily readable by other people.
6. One final note: Cheating irreversibly leads to the grade 1.

Exercise 1 – General Questions (18 points | approx. 18 minutes)

Answer the following questions:

a) Mention two different methods to implement a thread in Java. **(2 points)** **Answer:**

Subclassing the Thread class *and implementing the* Runnable interface. *For both approaches you need to put the logic into the run() method.*

b) Explain what a Java thread pool is in the context of the Java interface `Executor`, *and* list one advantage over manual thread management. **(2 points)** **Answer:**

Thread pools consist of worker threads which are automatically spawned and maintained by an implementation of the (interface) Executor.
Advantage: Developers don't have to do any thread management on their own, instead they can use the provided services by the Executor framework.

c) Explain *safety* in the context of concurrency, *and* provide an example that violates it. **(2 points)** **Answer:**

Safety refers to the correctness/integrity of the system, i.e., *corrupted values due to race conditions.*
Example: two threads concurrently write a value to the same variable which is not protected.

d) Explain *liveness* in the context of concurrency, _and_ provide an example that violates it. **(2 points)**
Answer:

Liveness refers to the progress of the system, i.e., *a thread is live when it can do eventually some progress.*
Example: Deadlock caused by circular dependencies.

e) How can a deadlock, starvation, or a race condition occur in the context of the *dining philosophers problem*? Explain each of them. **(3 points)** **Answer:**

Deadlock: Every philosopher wants to acquire the right fork and waits until another philosopher doesn't want to acquire the fork anymore
starvation: some philosophers always eat while others don't have the possibility to eat
race condition: two philosophers can take the same fork

f) When does the Java VM assign the three thread states NEW/RUNNING/WAITING to threads? **(3 points)** **Answer:**

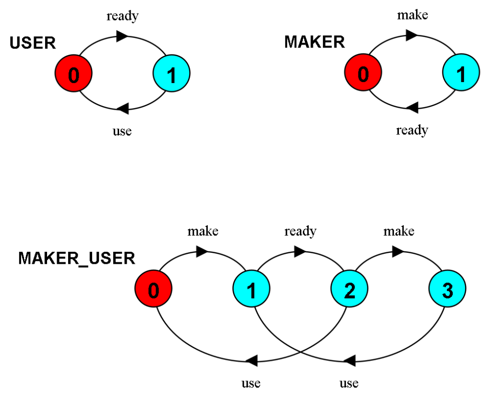
NEW: a new thread is initiated but not yet started
RUNNING: a thread is executing
WAITING: a thread called wait() and released its monitor

g) The Java language and its libraries already support many different synchronization techniques. Provide the names of four different Java classes that provide a ready to use synchronization techniques (only mention the class names, no explanation required). **(4 points)** **Answer:**

Object, Semaphore, CyclicBarrier, ReentrantLock, ... (not Lock, because that is an interface and not a class)

Exercise 2 – LTS (9 points | approx. 9 minutes)

Consider the FSP graphs below and translate the graphs to their corresponding LTS code. **(9 points)**



```
MAKER   = (                                          ).


USER    = (                                          ).


||MAKER_USER = (                    ).
```

Answer:

```
MAKER = ( make -> ready -> MAKER ).
USER = ( ready -> use -> USER ).
||MAKER_USER = ( MAKER || USER ).
```

Exercise 3 – FSP (12 points | approx. 12 minutes)

Consider the LTS below and answer the questions. The corresponding FSP graph is shown in Figure 1.

```
MAZE(Start=8) = P[Start],
P[0] = (north->STOP|east->P[1]),
P[1] = (east ->P[2]|south->P[4]|west->P[0]),
P[2] = (south->P[5]|west ->P[1]),
P[3] = (east ->P[4]|south->P[6]),
P[4] = (north->P[1]|west ->P[3]),
P[5] = (north->P[2]|south->P[8]),
P[6] = (north->P[3]),
P[7] = (east ->P[8]),
P[8] = (north->P[5]|west->P[7]).
||GETOUT = MAZE.
```

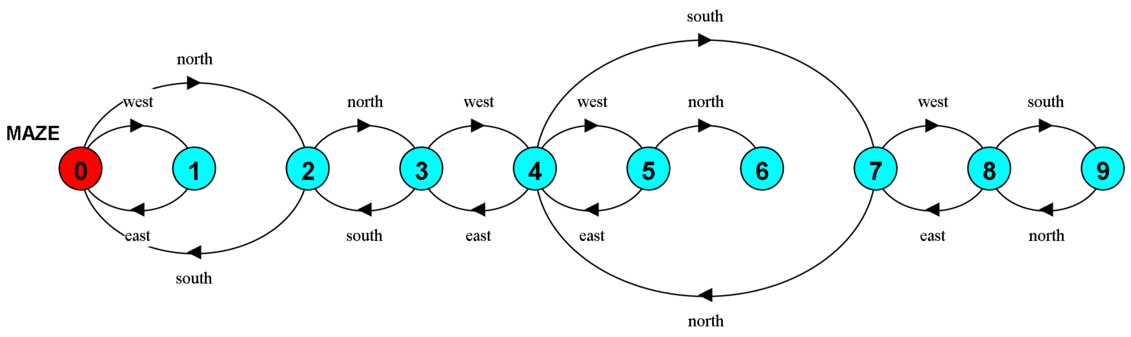


Figure 1: FSP graph for a maze

You will find the questions on the next page.

a) List the four *necessary and sufficient conditions* for deadlock occurrence. **(4 points)** **Answer:**

Serially reusable resources: the deadlocked processes share resources under mutual exclusion.
Incremental acquisition: processes hold on to acquired resources while waiting to obtain additional ones.
No pre-emption: once acquired by a process, resources cannot be pre-empted but only released voluntarily.
Wait-for cycle: a cycle of processes exists in which each process holds a resource which its successor in the cycle is waiting to acquire.

b) How many states does the FSP in Figure 1 have? Which is the starting state? Does it have a deadlock state? If yes, which one? **(3 points)** **Answer:**

10 states, 0 is the starting state, yes it has a deadlock state: 6

c) Provide two different terminal traces for the FSP in Figure 1. **(4 points)** **Answer:**

Trace 1: north, north, west, west, north
Trace 2: west, east, north, north, west, west, north
...

d) Is the number of possible traces in Figure 1 finite? Justify your decision! **(2 points)** **Answer:**
No, it's infinite because there are loops that can generate arbitrary traces.

Exercise 4 – Petri Net (15 points | approx. 15 minutes)

Please solve the following tasks regarding the Petri Net in Figure 2:

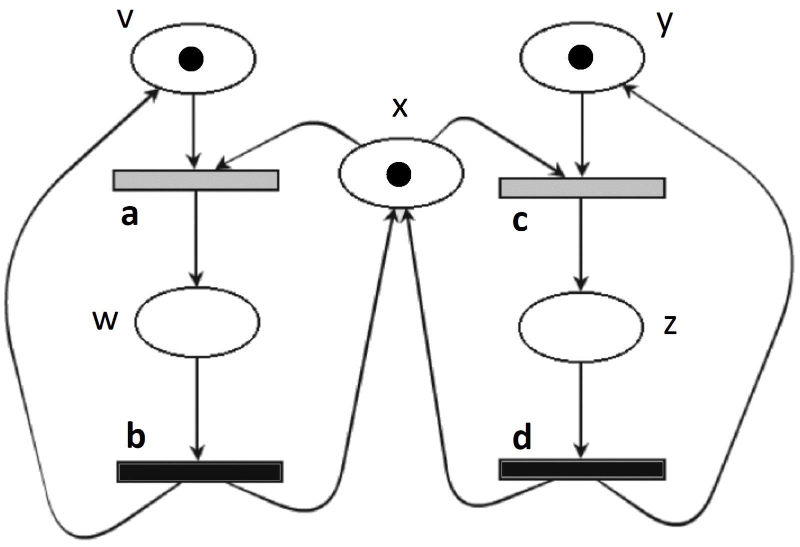


Figure 2: Mutual exclusion Petri Net

a) Complete the definition (*i.e.*, input functions, output functions, marking) of the Petri Net. **(3 points)**

P = {v, w, x, y, z}

T = {a, b, c, d}

I(a) = {__________}, **O(a)** = {__________}

I(b) = {__________}, **O(b)** = {__________}

I(c) = {__________}, **O(c)** = {__________}

I(d) = {__________}, **O(d)** = {__________}

m = {__________}

Answer:

I(a) = {*v*,*x*}, *O(a)* = {*w*}

I(b) = {*w*}, *O(b)* = {*v*,*x*}

$I(c) = \{x,y\}$, $O(c) = \{z\}$

$I(d) = \{z\}$, $O(d) = \{x,y\}$

$m = \{v, x, y\}$

b) Is the Petri Net bounded? Justify your decision! **(3 points)** **Answer:**

Yes, in the whole net there can be at most three tokens at a time.

c) Is the Petri Net safe? Justify your decision! **(3 points)** **Answer:**

Yes it is safe. The net can at most have one token per place.

d) Is the Petri Net conservative? Justify your decision! **(3 points)** **Answer:**

No, it's not conservative because the number of tokens is not constant.

e) Are all the transitions live in the Petri Net? Justify your decision! **(3 points)** **Answer:**

Yes, all transition are live because the mutex represents a repetitive process.

Exercise 5 – Practical Use of Concurrency (6 points | approx. 6 minutes)

Consider the code in Listing 1 and answer the following questions:

a) How many threads are spawned in the main process and what is the shared variable? **(2 points)** **Answer:**

5000 threads are spawned, list is the shared variable

b) Which major *safety* problem exists and what is the line number that causes that problem? **(2 points)** **Answer:**

Line 13: Use of a not thread-safe ArrayList causes corrupted data in the list, i.e., *it can happen that some values are not appropriately stored in the list. When we verify the size of the list after all threads finished, it is not 5000 as it should be, but sometimes only 4998 or even less.*

c) How could you mitigate the problem? Provide example code. **(2 points)** **Answer:**

Use a concurrent thread safe array or synchronize manually:
synchronized(list) { *list.add(threadName);* }

```java
public class ConcurrencyExample extends Thread {

    protected static ArrayList<String> list = new ArrayList<String>();

    @Override
    public void run() {
        String threadName = Thread.currentThread().getName();
        list.add(threadName);
    }

    public static void main(String[] args) {
        for (int i = 0; i < 5000; i++) {
            new ConcurrencyExample().start();
        }
    }
}
```

Listing 1: Sample Java code that uses threads