Concurrency:
State Models & Design Patterns

Practical Session

Week 06

Exercises 05

Discussion

Exercise 05 - Task 1

a) What is a guarded method and when should it be preferred over balking?

A guarded method is a pattern with the intent of temporarily suspending an incoming thread in case the object is not in the appropriate state to fulfill a request. Then it waits until the state changes. It should be used rather than when

- Clients tolerate indefinite postponement

- It is guaranteed that the required state is reached eventually

b) Why must you re-establish the class invariant before calling wait()?

When wait() is called the system releases the synchronization lock so state have to be consistent when you leave the method.

Exercise 05 - Task 1

c) What is, in your opinion, the best strategy to deal with an InterruptedException? Justify your answer!

Establish a policy to deal with InterruptedExceptions. Possibilities include:

- Ignore interrupts (i.e., an empty catch clause), which preserves safety at the possible expense of liveness.

- Terminate the current thread (stop). This preserves safety, though with brute-force! (Not recommended.)

…

d) How can you detect deadlock? How can you avoid it?

Trying to check for the presence of one of those sufficient condition for deadlock. To avoid deadlock you should design the system so that a waits-for cycle cannot possibly arise.

Exercise 05 - Task 1

e) Why it is generally a good idea to avoid deadlocks from the beginning instead of relying on deadlock detection techniques?

It is generally much harder to detect deadlocks in existing systems/programs than in models. Modeling a system during design allows to detect potential deadlocks right away and thus avoids to create deadlock-prone systems.

f) Why is progress a liveness rather than a safety issue?
A liveness property asserts that something good eventually happens. It means that the process have to have some progress in its executions. Safety is related to the state of the process, so a process could starve even if its state is in a correct state.

Exercise 05 - Task 1

g) Why should you usually prefer notifyAll() to notify()?

In most cases you do not know which process (thread) to wake up. So by using notify() you risk to wake up the wrong process and let other processes waiting.

h) What are the differences and similarities between a nested monitor lockout (a.k.a. nested deadlock) and a classical deadlock?

In a nested monitor lockout, Thread 1 is holding a lock A, and waits for a signal from Thread 2. Thread 2 needs the lock A to send the signal to Thread 1. Concludingly, in deadlocks both threads are waiting for each other releasing locks, while in nested deadlocks just one thread waits for the release. However, both of them lead to the same results, e.g. applications that block and potentially don't interact with any kind of inputs any longer.

Exercise 05 - Task 2

You have seen the dining philosophers problem during lecture. Please describe four different solutions to this deadlock problem and explain the concept and fairness characteristics of each.

1) Controller (fair)

2) Token Ring (fair)

3) Resource Ordering (not fair)

4) Queue (not fair)   *you will find the details in the solution PDF*

Exercise 05 - Task 3

Modify the FSP specification, in a different manner than seen in the lecture, to remove the deadlock. Fairness is not mandatory.

```
PHIL = ( sitdown
-> right.get -> left.get -> eat
-> left.put -> right.put -> arise -> PHIL ).
FORK = ( get -> put -> FORK).
...
```

```
PHIL(I=0) = (sitdown ->
   (when (I%2==0)
      right.get -> left.get
      -> eat -> left.put -> right.put
      -> arise -> PHIL
   | when (I%2==1)
      left.get -> right.get
      -> eat -> right.put -> left.put
      -> arise -> PHIL)
).
```

Exercise 05 - Task 4

Write a description of the maze as FSP process which, using deadlock analysis, finds the shortest path out of the maze starting at any square in the maze.

```
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(7).
```
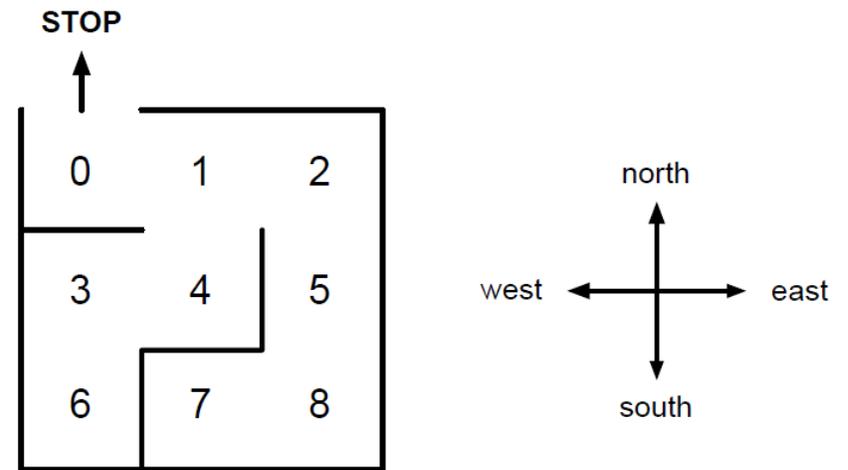
STOP

```
 0    1    2


 3    4    5


 6    7    8
```

north

west ←——→ east

south

Figure 1: A maze.

Lab 01

Details

Lab 01 - Overview

Concept

- Eclipse + Java based tasks

- Concurrency code that needs work

- Work in groups of two

- The lab starts at 10:15 and ends at 12:00

- Location will be <u>announced later through Piazza</u>

Grading

- 5 bonus points for 100% correctness

Requirements

- Notebook with power adapter

- Latest version of Eclipse installed

- Latest version of the Java SDK installed

- Optional: a mouse

Lab 01 - Procedure

1) Pull from public GitHub repository

   https://github.com/pgadient/concurrency_lab01.git

2) Fix the four provided sample apps

3) Submit your zipped project solutions by mail to
   gadient@inf.unibe.ch

Allowed:

- Lecture slides and books

- Internet access

Lab 01 - Solutions

Pull from public GitHub repository

https://github.com/pgadient/concurrency_lab01_solution.git

Please keep in mind that I will turn this repository for obvious reasons into a private one by the end of the day!

So please hurry to retrieve your own copy.