Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Leonel Merino

Solution
Series 05 — 18.10.2017 – v1.1a
Liveness and Guarded Methods

Exercise 1 (4 Points)

Answer the following questions:

a. What is a guarded method and when should it be preferred over balking? **Answer:**

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 balking (see serie 4, ex. 1) 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()? **Answer:**

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

c. What is, in your opinion, the best strategy to deal with an InterruptedException? Justify your answer! **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.)*
- *Exit the method, possibly raising an exception. This preserves liveness but may require the caller to take special action to preserve safety.*
- *Cleanup and restart.*

d. How can you detect deadlock? How can you avoid it? **Answer:**

Trying to check for the presence of one of those sufficient condition for deadlock.

- *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.*

To avoid deadlock you should design the system so that a waits-for cycle cannot possibly arise.

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

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? **Answer:**

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.

g. Why should you usually prefer notifyAll() to notify()? **Answer:**

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? **Answer:**

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 2 (2 Points)

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. **Answer:**

- *Controller: Before a philosopher can start to eat, he needs to ask the controller for permission. The philosopher who was hungry for the longest period of time gets prioritized. The requests are handled by a queue. As soon as a philosopher finished eating, he has to inform the controller about that. This approach is fair and resolves the deadlock problem. However, it is not very efficient, as just one philosopher can eat at a time.*
- *Token Ring: One philosopher after another can eat. In case the philosopher is not hungry, the others have still to wait until he gets hungry and finishes eating. There exist variations with multiple*

philosophers eating together at a time based on a fixed scheme, but then the philosophers that eat at the same time need to be selected carefully to mitigate potential deadlocks (neighbours mustn't grouped together in the scheme). This approach is very inefficient, but fair, and solves the deadlock problem.

- *Resource Ordering: The forks are the resources with an assigned priority. Each fork therefore gets assigned a unique number. As soon as a philosopher wants to eat, he needs to take the fork with the smaller number first, as long as it's free. Next, he's allowed to grab the fork with the larger number. With this solution at least one philosopher can eat at any time, but it's not fair, because it is not guaranteed that every philosopher will be able to take his forks. However, it's free of deadlocks.*
- *Queue: Whenever a philosopher wants to eat, his request gets queued in a queue as long as he's not able to grab both forks at a time. This is the case when at least one fork is already occupied by another philosopher. As soon as he can grab both forks at a time, he starts eating and releases both forks after he finished his meal. Then, the next philosopher in the queue is potentially allowed to eat. This solution is not fair, because it doesn't make any guarantees that every philosopher will be able to eat eventually, but at least it solves the deadlock issue.*

Exercise 3 (2 Points)

Consider the FSP model for the dining philosphers:

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

FORK = ( get -> put -> FORK).

||DINERS(N=5) =
forall [i:0..N-1]( phil[i]:PHIL
|| {phil[i].left,phil[((i-1)+N)%N].right}::FORK ).
```

Modify the FSP specification, in a different manner than seen in the lecture, to remove the deadlock. Fairness is not mandatory.
Hint: You can search for deadlocks with the LTSA tool through the menubar at "Check" - "Safety" or "Check" - "Progress". **Answer:**

```
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)
).

FORK = ( get -> put -> FORK).

||DINERS(N=5) = forall [i:0..N-1]
      ( phil[i]:PHIL(i) || {phil[i].left,phil[((i+N)-1)%N].right}::FORK ).
```

Exercise 4 (2 Points)

In a FSP model of a process a deadlocked state is a state with no outgoing transitions, a terminal state so to say. A process in such a state can engage in no further actions. In FSP, this deadlocked state is represented by the local process STOP. By performing a breadth-first search of the LTS graph (in CHECK PROGRESS), the LTSA tool guarantees that a sample trace is the shortest trace to the deadlock state.

Consider the maze depicted in Figure 1. 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.

You may check your solution by inspecting the *trace to deadlock* from specific squares.

Hint: At each numbered square in the maze, a directional action can be used to indicate an allowed path to another square.

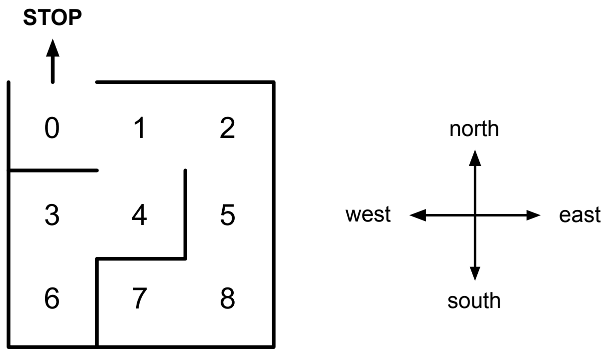

Figure 1: A maze.

Answer:

```
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).
```