Prof. Oscar Nierstrasz
Haidar Osman, Leonel Merino, Nevena Milojković

Solution Serie 5 - Liveness and Guarded Methods

Exercise 1

Answer the following questions: (Each answer 0.5 points)

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 right 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 brutally! (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.

Exercise 2

(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.
Answer:

```
// not fair solution:

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 3

(2 points)

Exercise 4

(3 points)