Marcel ZAUDER 16-124-836
Pascal GERIG 16-104-721

5.1 Several Questions

a) *When should a guarded method be preferred over balking?*

b) *What is in your opinion the best strategy to deal with an InterruptedException? Justify your answer!*
One way to handle an InterruptedException is to propagate it to the caller so it can determine what should be done when such a situation occurs. Unfortunately when the code is part of a Runnable, which will most often be the case, nothing can be thrown, therefore the status should be restored.
In my opinion one should use a custom exception handling, such that for every circumstance an individually fitted handling can be created. For instance, when a thread is waiting for an I/O-operation and is interrupted any resource that is held by it can be closed before the thread is terminated. Handling such exceptions correctly and in a predefined way fitting to the circumstances can balance the responsiveness and robustness of the application.

c) *How can you detect a deadlock?*
One way to detect deadlocks is to create threads that create checkpoints. For example when a thread is in a working loop, one can create a timer which is set to a longer time than a thread should need to accomplish the work. If the timer hits zero the checkpoint thread assumes that the thread in question is deadlocked.

d) *How can you avoid a deadlock?*
One of the simplest way to avoid deadlock would be to create your program/algorithm such that it works sequentially, without concurrency a deadlock cannot occur.
If one wants to use concurrency and avoid deadlocks there are four possible ways to do so:
First we can make sure to not use serially reuseable resources so if several shared resources shall be taken they should either be taken simultaneously or the next one should only be taken if the first one was already acquired.
Another way is to not let process hold on there resources they acquired before while they wait, so if they reach a "wait-cycle" they should release their access of the resources they already have.
A third way is to use pre-emption such that when a process needs a resource that is currently held by another one, it should be able to ask the process to give the resource to it and then return it after finishing its actions.
A last way is to avoid wait-for-cycles, so no process in a cycle would wait for a resource currently being held by its successor.

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

f) *Why is progress a liveness rather than a safety issue?*
Liveness tells us that (eventually) something good will happen, therefore it ensures that *progress* is being made. A program can also make progress while being in a wrong state, so it cannot check if during the hold trace sequence something bad has happened but will only check if the liveness property can be reached at any time. Therefore progress is not a safety issue.

g) *Why should you usually prefer notifyAll() to notify()?*
One should use *notifyAll()* preferably to *notify()*, because the later one will only wake up one randomly selected threadd that is waiting for an object. Therefore, reffering to the BoundedCounter example from the lecture, if we have two threads waiting for the BoundedCounter resource to be released - one wants to decrease one wants to increase the counter, and the last thread that was working on it decreased the counter to 0 and then notifies with *notify* it can happen that the thread that wants to decrease the counter is woken up, but because the counter is 0 it will wait again because the BoundedCounter cannot be decreased below 0. Then no thread will enter the *Critical Section* and therefore no progress will be made. If we used instead the *notifyAll()* method the "Increaser" thread could work on its action so the counter is increased.

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

In a nested monitor deadlock only one thread is part of the deadlock, because it tries to acquire the monitor that it is already holding and therefore somewhat waits for itself. In a classical deadlock at least two or more threads are waiting for each other to release their resources in order to be able to execute their actions.

5.2 Dining Philosophers - Deadlock Avoidance IDEAS

1. *One Philosopher takes forks in reverse order*
   We can create an implementation in which at least one philosopher, instead of first acquiring his right fork and then the left, acquires first his left fork and then the right. In regards to fairness it can happen that if this philosopher is very slow to acquire the forks it can starve because the other philosophers are much faster to acquire the forks.
2. *Only n-1 philosophers are allowed at the table*
   When using a queue before the philosophers are sitting down at the table and only letting n-1 philosophers at most sit at the table, we would also avoid a deadlock because there are enough forks such that at least one philosopher can eat. Because there is a FIFO queue beforehand it is also a fairer improvement but it can happen that one philosopher is sitting at the table, already acquired a fork, but because he is really slow and the other philosophers are much faster at acquiring the other forks he can also starve, but this is less of a problem than in the example before.
3. *Philosopher can talk with each other and ask for fork (Pre-Emption)*
   In this improvenment example the philosophers can talk with the philosopher next to him, if they want to acquire the fork already held by that philosopher in order to ask to get access for it. In regards to fairness this is a pretty good solution similar to the second one, because one philosopher will only starve if he is really slow and other philosophers are always taking his already acquired fork away.
4. *No idea*

5.3 Dining Philosophers - Deadlock Avoidance FSP/LTSA

The idea here is that each philosopher with an odd ID first grabs his left fork and then the right fork and each philosopher with an even ID first grabs his right fork and then the left fork and therefore this implementation avoids a deadlock:

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

PHIL2 = ( sitdown
-> left.get -> right.get -> eat
-> right.put -> left.put -> arise -> PHIL2 ).

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

||DINERS(N=2) = forall [i:0..N-1](
        phil[i*2]:PHIL || {phil[i*2].left,phil[(((i*2)-1)+(2*N))%(2*N)].right}::FORK ||
        phil[i*2+1]:PHIL2 || {phil[i*2+1].left,phil[(((i*2+1)-1)+(2*N))%(2*N)].right}::FORK
        ).
```

5.4 Maze

```
const STARTTILE = 6
```

Marcel ZAUDER 16-124-836
Pascal GERIG 16-104-721

```
Maze = TILE[STARTTILE],
TILE[0] = (north -> STOP | east -> TILE[1]),
TILE[1] = (east -> TILE[2] | south -> TILE[4] | west -> TILE[0]),
TILE[2] = (west -> TILE[1] | south -> TILE[5]),
TILE[3] = (east -> TILE[4] | south -> TILE[6]),
TILE[4] = (north -> TILE[1] | west -> TILE[3]),
TILE[5] = (north -> TILE[2] | south -> TILE[8]),
TILE[6] = (north -> TILE[3]),
TILE[7] = (east -> TILE[8]),
TILE[8] = (north -> TILE[5] | west -> TILE[7]).
```