Solution Exercise Set 8 - Asynchrony & Condition Objects

Exercise 1

Answer the following questions: (0.5 points each)

a) Why are servers (e.g. web servers) usually structured as thread-per-message gateways? **Answer:**

Servers (e.g. web servers) usually must handle many requests, some of them quite resource-intensive. It makes then sense to let each request (service) to be handled by a separate helper thread, as long as the overhead of thread construction does not exceed the helper costs. Load balancing can be realized by distributing different tasks among separate threads or processes.

b) What are condition objects?. Name an advantage and a disadvantage of using condition objects. **Answer:**

Condition objects are a programming pattern used to encapsulate waiting and notification strategies in the context of guarded methods.
Advantages: (1) more complicated synchronization strategies or scheduling policies can be sourced out to simplify class design, (2) by isolating conditions, one can often avoid notifying waiting threads that possibly cannot proceed given a particular state change (efficiency).
Disadavantages: (1) the first advantage may turn into a disadvantage: java language constraints might increase design complexity (e.g. see solution of bounded counter example in the lecture), (2) may lead to "design overhead" in some situations, (3) if used without care, may easily lead to nested monitor problem (but this applies to synchronization mechanisms in general..)

c) Why does the SimpleConditionObject from the lecture not need any instance variables? **Answer:**

The Condition class does not need to implement any state, rather it specifies synchronization behaviour. Simple condition objects are then used as "synchronization variables" by other classes (e.g. bounded counter) that need synchronization.

d) What is the "nested monitor problem"?. Give an answer that is as precise and short as possible (in your own words). **Answer:**

The nested monitor problem is a synchronization problem leading to a deadlock. It typically occurs when two or more levels of synchronization are used: a blocked thread holds a wait on a resource that contains a method which could signal to unblock the lock, but since the signal never occurs, all threads block each other. For an example, see the bounded buffer from the lecture, or Ex. 3.

e) What are "permits" and "latches"?. When it is natural to use them? **Answer:**

Basically, permits are special condition objects that behave similarly to counting semaphores. A permit class maintains a permit counter representing the number of available "permits" (e.g. number of allowed objects in bounded buffer, number of possible increments in bounded counter, etc.). In fact, one could say that a permit is a counting semaphore implemented as condition object. The signal operation increments, and the await operation decrements the permit counter. A latch is a condition that is initially false, but once set to true, it remains true. See for example the `CountDownLatch` *class in* `java.util.concurrent`.

Such kind of mechanisms are useful whenever synchronization depends on the value of some counter.

Exercise 2: Futures (2.5 points)

Consider the sample code FutureTaskDemo.java and EarlyReplyDemo.java which you both find in CP-Series8.zip [1] in the package asynchrony. In both cases, several client threads request a server to compute fibonacci numbers.

a) Which implementation would you prefer for this kind of problem?. Is there any considerable difference at all?. Justify your answer! **Answer:**

In the early-reply approach, the client waits for a service (which has been delegated by the host to a helper) until the host gets the result of the computation (the host retains synchronization). In the future-task approach, the client receives from the host a future object, which represents the (future) result of the (asynchronous) computation. This computation executes in parallel while the client in fact may continue its work.
In our example, every demoThread receives a future object, and depending on the exact scheduling, the clients waiting for the faster computations are served first. In the early reply example, all clients are waiting for the slowest computation, if this was the first request. It depends much on the order of requests which clients are served first. So the future-based solution seems more flexible in this case.

b) Write a new class FutureTaskExecDemo.java that uses an executor service to compute the future task and to execute the clients, instead of creating explicit new threads. What is the benefit of using executors? **Answer:**

The task is usually submitted to a pool of threads of which one will execute your task. This is convenient because you don't have to create explicitly new threads, and the code that submits the task does not have to be aware of which thread exactly executes the task.

c) In addition, add a time constraint such that the client thread waits for at most a given amount of time for the result. **Answer:**

The solution is found in FutureTaskExecDemo.java in the CP-Series8-Sol.zip folder.

Exercise 3: Nested Monitor (2 points)

Farmer Napoleon owns a magic chicken called Clarissa who is supposed to lay infinitely many eggs. Napoleon has hoped to dispose of an endless source of eggs to build up his egg-imperium. But there is a serious deadlock problem hidden behind the story. Your task is to resolve this problem in order to let Napoleon to continously retrieve eggs from Clarissa. Be careful that your solution is data race free.

You find the code in CP-Series8.zip [1] in the package eggFarm.

[1] http://scg.unibe.ch/download/lectures/cp-exercises-2015/CP-Series8.zip

Answer:

The solutions can be found in CP-Series8-Sol.zip [2].

Exercise 4: Nested Monitor revisited (3 points)

Write an LTS model for the modified program from Exercise 3. Make sure that no deadlock occurs; check your model using the LTSA tool. Hint: You should model at least the following processes: `SEMAPHORE`, `THENEST`, `CHICKEN`, `FARMER`, and the composite process `FARM`.

Answer:

```
SEMAPHORE(I=0) = SEMA[I],
SEMA[v:0..5] = (when (v<5) up->SEMA[v+1]
| when (v>0) down->SEMA[v-1]
).

THENEST = (put ->THENEST | get -> THENEST ).

CHICKEN = (empty.down -> put -> full.up -> CHICKEN).
FARMER = (full.down -> get -> empty.up -> FARMER).

||FARM = (CHICKEN|| THENEST || FARMER
||empty:SEMAPHORE(5)
||full:SEMAPHORE(0))@{put,get}.
```

[2] http://scg.unibe.ch/download/lectures/cp-exercises-2015/CP-Series8-Sol.zip