Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Leonel Merino

Solution
Series 01 — 20.09.2017 – v1.0a
Introduction to Concurrency

Exercise 1 (7 Points)

Answer the following questions (1 point each):

a. Do recent central processing units (CPUs) of desktop PCs support concurrency? Why became concurrency for many software applications very important these days? **Answer:**

Yes, they do. Not only can recent CPUs handle multiple threads on a single core (preemptive multitasking), they also leverage multiple processing cores to increase overall performance. In todays connected world large-scale data analysis requesting huge compute clusters exploiting massive concurrency became very popular in machine learning or trend prediction.

b. What is safety? Give one concrete example of a safety violation. **Answer:**

Safety is a property of concurrent programs and it claims that the consistency of resources is ensured, i.e., data cannot be corrupted by concurrent processes. In other words, "nothing bad happens". For example, a safety violation occurs when two bank transactions happen at the very same time, and the bank account ends up with the wrong balance, respectively in the wrong state.

c. What is liveness? Give a concrete example of a liveness violation. **Answer:**

Liveness is a property of concurrent programs and states that processes are eventually (i.e. guaranteed at some time in the future) able to do useful work, e.g., read or write data. In other words, "something good eventually happens". A liveness violation occurs, if someone makes a bank transaction and the transaction never finishes.

d. Using the implementation in the slides, can a binary semaphore lead to a deadlock? Can it lead to starvation? Explain with the aid of an example. **Answer:**

The binary semaphore cannot lead to a deadlock. However, depending on the implementation, it may or may not lead to starvation. If too many processes access the critical section, an unfair implementation of the semaphore can starve some processes.

e. Why do we need synchronization mechanisms in concurrent programs? **Answer:**

Synchronization is needed to ensure safety and liveness in concurrent programs. If processes do not need to communicate, or do not share a common state, then synchronization is unnecessary. This sort of concurrent programs involve "embarrassingly parallel problems" that can easily be parallelized without any synchronization constructs. As soon as processes need to communicate, synchronization is mandatory to make sure "good things happen", in the right order, producing the right outcome.

f. How do monitors differ from semaphores? Please provide a precise answer. **Answer:**

They differ in terms of flexibility, usage, and programming style. Semaphores can allow several concurrent processes to enter the mutex, whereas with monitors it is just one. Further, with monitors the signal statement to wake up waiting processes is more limited than with semaphores, i.e., a developer can call the signal method multiple times without any potential side effects (apart from spending much time to other processes), whereas with semaphores each time V(s) gets called an additional processes is potentially allowed to enter the mutex. For example, one semaphore initialized with 3 allows developers to establish 3 *concurrent processes to be in the mutex domain, whereas with each monitor just up to one process can reside in that domain.*

g. How are monitors and message passing similar? And how are they different? **Answer:**

Monitors and message passing are similar in the sense that they both encapsulate data and operations. They are different in the sense that monitors rely on shared data and processes need to sync using locks, whereas message passing uses no shared state and messages are used to achieve synchronization.

Exercise 2 (2 points)

```
x := 1
Thread 1 -> x := x + 7.
Thread 2 -> x := x * 5.
```

Considering the previous code, give all possible values of x at the end of the execution of both threads with their corresponding execution traces.

Hint: You should be able to perceive 6 different execution flows in total, however, some of them could may lead to the same x.

Answer:

Every thread has a read (r) operation and a write operation (w). Let's list all the possible combinations (read must occur before write in each process) and see the results:

```
r1(x = 1), w1(x = 8), r2(x = 8), w2(x = 40)  -->  40
r1(x = 1), r2(x = 1), w1(x = 8), w2(x = 5)   -->  5
r1(x = 1), r2(x = 1), w2(x = 5), w1(x = 8)   -->  8
r2(x = 1), r1(x = 1), w2(x = 5), w1(x = 8)   -->  8
r2(x = 1), r1(x = 1), w1(x = 8), w2(x = 5)   -->  5
r2(x = 1), w2(x = 5), r1(x = 5), w1(x = 12)  -->  12
```

Exercise 3 (1 points)

Implement a monitor using semaphores. Use pseudo-code and comment it.
Answer:

A monitor is a combined data structure along with its operations. At most one process can be in the monitor at any given time. To implement a monitor we have to provide a lock to lock the monitor and a mechanism to signal and wait. The general strategy is to implement a lock based on a semaphore and then implement a monitor based on the lock.

```
type Lock{
  BinarySemaphore bs=1;
}

acquire(lock:Lock){
  P(lock.bs)
}
```

```
release(lock:Lock){
   V(lock.bs)
}

type Monitor{
   Lock lock;          //This is used to ensure mutual exclusion of the
                         waiting processes
   BinarySemaphore bs=0;
   wait(){
      release(lock);   //Release the lock of this monitor allowing other
                         processes to acquire it
      P(bs);           //Make this process wait for the signal (reschedule
                         the process)
      acquire(lock);   //Acquire the lock on wake up
   }

   signal(){
      V(bs)
   }
}
```

The condition variable can be used as in the lecture slides. Note that in method wait()*, the release of the lock and the waiting should happen atomically (this can be implemented by yet another semaphore or lock). For a data type and its operations to be a monitor, every operation should start by acquiring the lock and end by releasing the lock. The same lock should be used in any condition variable within the monitor. Remember that every* wait *should have a* signal *somewhere in the operations. This is very similar to how* wait *and* notify *work in Java. Finally, keep in mind that this is a rough implementation just to give the idea how we can build a concurrency concept based on another.*