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

Solution Serie 1 - Introduction to Concurrent Programming — 16.09.2015

Exercise 1 (6 Points)

Answer the following questions (1 point each):

1. What is safety? Give one concrete example of a safety violation.
2. What is liveness? Give a concrete example of a liveness violation. **Answer:**

   Safety and liveness are properties of concurrent programs. A safety property states that consistency of resources is ensured: i.e. data cannot be corrupted by concurrent processes. In other words, "nothing bad happens". Liveness means that processes are eventually able to do useful work, i.e read or write data. In other words, "Something good eventually happens". An example of a safety violation is when tow bank transaction happens at the same time and the bank account ends up in the wrong state. A liveness violation example would be if I make a bank transaction and the transaction never finishes.

3. 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.

4. 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, do not share code, or do not share state, then synchronization is not needed. These are called the "Embarrassingly parallel problems". The moment processes need to communicate, synchronization is needed to make sure good things happen, in the right order, producing the right outcome.

5. How exactly do monitors differ from semaphores? **Answer:**

   Semaphores are basic synchronization concepts. They provide a simple locking mechanism. Monitors ... (see the last question)

6. 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 assume shared data and processes need to sync using locks, whereas message passing assume no shared state and messages are used to achieve synchronization.

Exercise 2 (2 points)

```
x := 1
Thread 1 -> x := x + 5.
Thread 2 -> x := x * 3.
```

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

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

```
r1(x = 1), w1(x = 6), r2(x = 6), w2(x = 18)  ---> 18
r1(x = 1), r2(x = 1), w1(x = 6), w2(x = 3)  --->  3
r1(x = 1), r2(x = 1), w2(x = 3), w1(x = 6)  --->  6
r2(x = 1), r1(x = 1), w2(x = 3), w1(x = 6)  --->  6
r2(x = 1), r1(x = 1), w1(x = 6), w2(x = 3)  --->  3
r2(x = 1), w2(x = 3), r1(x = 3), w1(x = 8)  --->  8
```

Exercise 3

Implement a monitor using semaphores. Use pseudo-code and comment it. 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 condition variable based on the lock.

```
type Lock{
   BinarySemaphore bs=1;
}
acquire(lock:Lock){
      P(lock.bs)
}
release(lock:Lock){
      V(lock.bs)
}

type ConditionVariable{
   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 waking 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.