Prof. Dr. Oscar Nierstrasz
Pascal Gadient, Mohammadreza Hazhirpasand

Solution
Assignment 01 — 18.09.2019 – v1.1
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 today's connected world, large-scale data analysis requesting huge compute clusters exploiting massive concurrency became very popular in machine learning or trend prediction.

b) 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.

c) 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.*

d) 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.*

e) Can a binary semaphore lead to a deadlock? Why? Can it lead to starvation? Why? **Answer:**

The binary semaphore cannot lead to a deadlock, because it does not allow circular dependencies. However, depending on the implementation, it can lead to starvation due to processes that are not receiving any signallings.

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 mutual exclusion (mutex, also known as "critical section"), whereas with monitors it is just one. Further, with monitors the signal statement to wake up waiting processes is more limited than with semaphores. That is, 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 process will be allowed to enter the mutex. For example, a semaphore initialized with three tokens allows developers to run up to three concurrent processes within the critical section.

g) How are monitors and message passing similar? 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 code above: Give all possible values of `x` at the end of the execution of both threads together with their corresponding execution traces.

Hint: You should be able to perceive 6 different execution flows in total, however, some of them could 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 point)

Implement a monitor using a binary semaphore, *i.e.*, a semaphore that can contain at most one token. Use pseudo-code and comment it.

Answer:

At most one process is allowed to be in the monitor's critical section at any given time. Hence, a monitor requires a mutual exclusion (mutex) provider, i.e., *the lock. In addition, a monitor provides the accessor methods wait and signal. The idea is to start implementing a lock based on a binary semaphore and then continue with a monitor which uses that lock.*

```
class Lock {
  BinarySemaphore s = 1;  //1 allows the first process to enter the mutex.

  acquire() {
    this.s.p()            //Takes a token from the semaphore if available,
  }                         blocks otherwise.

  release() {
    this.s.v()            //Puts a token back to the semaphore.
  }
}
```

```
class Monitor {
  Lock lock;              //Ensures mutual exclusion of the waiting
                            processes.

  wait() {
    this.lock.acquire(); //Acquires the lock.  Forces a process to wait
                            if no tokens are available.
                            (rescheduling of the process)
    *do critical work here (avoid interruptions/ process terminations!)*
  }

  signal() {
    this.lock.release()  //Releases the lock of this monitor allowing
                            other processes to acquire it.
  }
}
```

Note that in method wait() *the waiting and signalling must happen atomically. This atomicity requires hardware guarantees,* e.g., *the CPU must support atomic hardware instructions. For a monitor, every wait() block should start with acquiring the lock. Within wait(), the same lock should be used for each process. Generally speaking, remember that every* wait *should be ultimately followed by a* signal *to avoid starvation due to missed wake up calls.*
NB: This concept is very similar to how wait *and* notify *work in Java. Please keep in mind that this is a rough implementation just to give you an idea of how to implement one concurrency concept with another.*