Joel Krebs, 09-112-996

Aaron Karper, 08-915-894

Concurrent Programming

Series 1

Exercise 1

What is the difference between concurrency, concurrent programming, and parallelism?
Concurrency is a property of a system where multiple computations and processes may execute simultaneously and access shared resources. Concurrent programming keeps this in mind and aims to solve problems which occur on such systems. Concurrent programs may, but not necessarily have to run in parallel. Parallelism on the other-hand is a state during runtime at which at least two processes run at the same time. Tasks and computations are split up in smaller ones which then can be distributed.

What are safety and liveness?
Safety is a principle in concurrent programming that ensures, that nothing bad happens. E.g. that you don't get corrupted data. Liveness on the other hand ensures that there is actual progress happening and no process starve or are locked out.

What is the difference between deadlock and starvation?
In a deadlock several actors are waiting on each other to finish. But because every actor is waiting, no actual progress can be made. Starvation happens to a specific process is unable to gain access to a required resources, usually due to malfunctioning scheduling of the resource.

Give an example of deadlock. (Technical example or invented situation in the real life)
We have two bank accounts A and B. We also have two threads: The first wants to transfer money from A to B and the second from B to A. The method to transfer money looks as follows:

```
transfer(Account from, Account to, Double amount) {
      lock(from)
      lock(to)
      from.withdraw(amount)
      to.deposit(amount)
      release(to)
      release(from)
}
```

Now in line 2 both threads are locking two resources they need. Thread 1 locks A and Thread 2 locks B. Then they're waiting for the second resource which just has been locked by the other thread. But as they're both waiting, they will never be releases.

Give an example of starvation. (Technical example or invented situation in the real life)
The classic fork-bomb: We have some process P that is useful to us and we want it to use RAM and CPU time. However, someone executes a process `B := spawn B; spawn B` on the same machine. This leads to a potentially infinite amount of processes, that will take the resources away from P up to the point, where P can't meaningfully continue to run. Note that B runs just fine and therefore this isn't a dead-lock.

Why do we need synchronization mechanisms in concurrent programs?
Many applications assume that certain complex parts are in fact atomic, in that no other process can edit or access the same resources as the current thread. If a printer would not block during printing, two pages would become mangled or bank accounts would show very surprising balances, because the transactions would not be atomic.

How exactly do monitors differ from semaphores?
The monitor ensures locking and releasing in the same method, semaphores could be P'ved and V'ved in different functions and there is no guarantee that it is used correctly.

When does it make sense to use busy-waiting?

If there is no other mechanism available or if there is no way for other processes to know if the complex condition on which I'm waiting is fulfilled.

Are binary semaphores as good as counting semaphores? Explain your answer.
This of course depends on the definition of "as good as". They can be used to implement counting semaphores:

```
with lock(counting) BLOCK :=
with lock(binary_1) {
      while (available == 0) { skip };
      available--;
      BLOCK
      available++;
}
```

but then again, counting semaphores have this management in a simpler manner, so binary isn't as good as counting, even if they are computationally equivalent.

What problems could nested monitors cause?
If not carefully executed, they might end up waiting for each other, causing a dead lock.

Exercise 2

6, 3, 18, 8 (and 1, in case of a deadlock)

Joel Krebs, 09-112-996

Aaron Karper, 08-915-894

Exercise 3

```
class Condition {
      private Semaphore c;

      public wait() {
            acquire(c)
            acquire(monitor's semaphore) // start executing a method.
      }
      public signal() {
            release(monitor's semaphore) // other methods are allowed
            release(c)
      }
}

class Monitor {
      private Semaphore lock

      public X someMethod() {
            acquire lock

            ...
            release lock
      }
}
```

Exercise 4

```
class Semaphore extends Monitor {
      private Condition nonZero
      private int count
      public void V() {
            count++
            nonZero.signal
      }

      public void P() {
            while (count == 0) {
                  nonZero.wait
            }
            count--
      }
}
```