Marcel Zauder 16-124-836
Pascal Gerig 16-104-721

1.1 Several Questions

a) *Do recent central processing units (CPUs) of Desktop PCs support concurrency?*
Yes the current architecture of multi-core processors does have the need to allow concurrency to maximize the power of the CPU.

b) *Why do we need synchronization mechanisms in concurrent programming?*
To reassure that no safety properties are violated, i.e. two threads are concurrently onto a savespace and therefore its value gets either corrupted or is false.

c) *What is safety in the context of concurrent programs?*
The safety property ensures that "nothing bad will happen" during the execution of a critical part of a program, i.e. unexpected or corrupted results should not occur.

d) *Give one concrete example for a safety violation.*
All executions of Exercise 2 in which the result is neither 8 nor 5 (concurrent executions 1 to 4).

e) *What is liveness in the contect of concurrent programs?*
The liveness property ensure that "(eventually) something good will happen" during the execution of a program.

f) *Give a concrete example of a liveness violation.*
A violation of a liveness property would be a deadlock in which one thread is waiting for a resource to be released to continue its work but the other thread is also waiting for the first one to release its resource to continue its work and therefore both threads are waiting for each other and no progress is made.

g) *Why or why not can a binary semaphore lead to a deadlock?*
When a thread that acquired a binary semaphore crashed it cannot release it and therefore the other threads that are waiting for the semaphore to be released are deadlocked.

h) *Why or why not can a binary semaphore lead to starvation?*
When a thread acquires a binary semaphore, executes the critical section, releases the semaphore but directly after - because thread 1 might be much faster than thread 2 - immediately reacquires the semaphore to execute the critical section again and this "ad infinum" thread 2 will "starve" because it is too slow to acquire the semaphore.

i) *How do monitors differ from semaphores? Please provide a precise answer.*
A semaphore is an integer variable that performs wait and signal operations, whereas a monitor is an abstract data type that allows only one process to use the shared resource at a time. A semaphore can be acquired by multiple threads/processes while a monitor ensures mutual exclusion.

j) *What are similarities between monitors and message passing?*
In (buffered) message passing and a monitor both use signaling to imply that a condition has been met. When using a monitor a thread does not need to care about synchronization similar to asynchronous message passing.

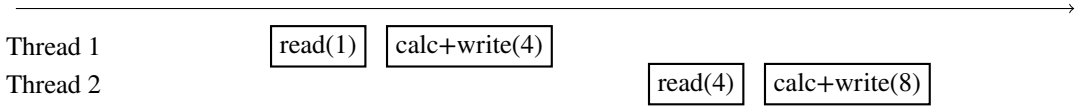
k) *What are the differences between monitors and message passing?*
A monitor blocks other threads from accessing a resource which is already held by a certain thread whereas in buffered message passing the "consumer thread" is first asking with a *signal* message if there is a value to take and if there is the buffer will send the value to the consumer requesting it with a synchronous *take* message.

Marcel Zauder 16-124-836
Pascal Gerig 16-104-721

1.2 Sequential/Concurrent Thread Executions

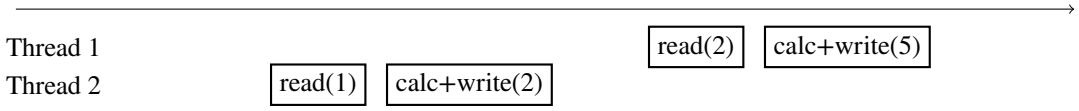
Assuming that reading and calculation + writing are in themself atomic.

1.2.1 1. Sequential Execution



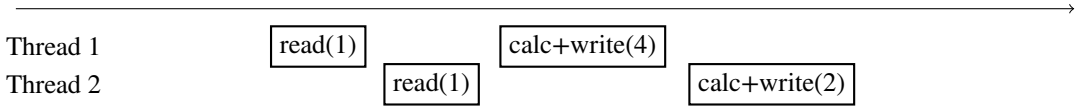
The result with this execution is **8**.

1.2.2 2. Sequential Execution



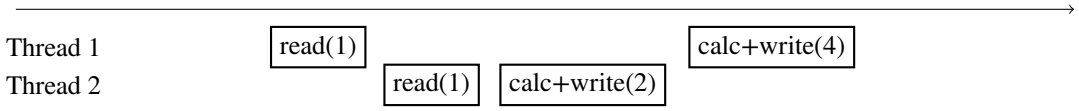
The result with this execution is **5**.

1.2.3 1. Concurrent Execution



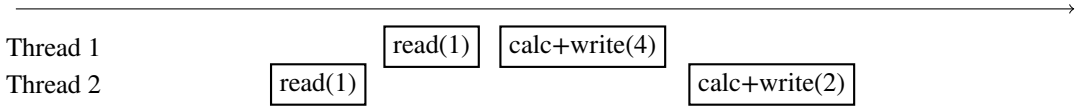
The result with this execution is **2**.

1.2.4 2. Concurrent Execution



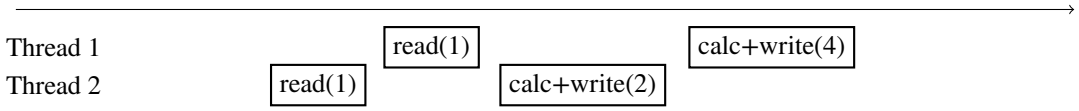
The result with this execution is **4**.

1.2.5 3. Concurrent Execution



The result with this execution is **2**.

1.2.6 4. Concurrent Execution



The result with this execution is **4**.

Marcel Zauder 16-124-836
Pascal Gerig 16-104-721

1.3 Monitor from a Binary Semaphore

```
private final Semaphore semaphore = new Semaphore(1)

public void monitor_wrap(Function f) {
        semaphore.acquire()
        try {
                execute f
        } finally { semaphore.release() }
}
```

Every method call in the monitor should call the monitor_wrap method with parameter of the method itself, so it is wrapped with a semaphore acquisition and thus mutex is guaranteed.