Operating Systems
Assignment 5.1 - Content Questions

Henri Heyden, Nike Pulow
stu240825, stu239549

5.1.1

Why can we not simply use time or clocks as means for synchronization?

Synchronization via time would usually result in slower processes, as for every process there must be a specific amount of time specified, in which this process needs to complete their task. This would also mean that if the process finishes its task way before, it still would have to wait for the timeout to occur. On the other hand, if a process needs more time than specified in its timeout interval, this might result in processes never being able to finish.
The same line of arguments goes for clocks - clocks are a great way to synchronize with real world time; yet, as mentioned above, time is not a great way for synchronization. Logical clocks on the other hand provide some control over the order of events in a distributed system or pose as useful checkpoints in alogrithms.
In summery: semaphores provide more controlled access to critical sections and shared resources, which of course is the whole point of synchronization in this context. While time might be able to do the same thing, it is just way more efficient to use semaphores, as there is less waste of resources because of waiting times.

5.1.2

What are deadlocks?

”Deadlock is the situation in which one or more processes within a system are blocked forever due to requirements which can never be satisfied.”
(Holt, 1971, p. 1)
In other words: In a deadlock situation a process A is waiting for a process B to release a resource, while process B is waiting on process A to release a resource.

What are livelocks?

A livelock occurs when ”one or more threads continuously change their states in response to changes in states of the other threads without doing any useful work.”
(Ganai, 2013, p. 1)

What is the difference and are they both equally difficult to resolve?

The main difference lies in what causes a live-/deadlock to occur: for deadlocks it is dependencies, for livelocks it is a status change in response to another status change.
Livelocks are harder to recognise than deadlocks, since the program counter of the involved processes is constantly changing, due to the status changes of the processes. There are several different strategies to solve both problems, all of which are dependent on circumstance. Usually, the deadlock problem is supposed to be handled by the application, whereas prevention is practised in real-time operation, while detection is the way to go in debugging situations.
Livelocks are usually being resolved by changing the decision-making logic (refer to the philosopher's problem) or by introducing randomness and delays.
If detected and analyzed properly, both problems are equally difficult to solve, just in different ways.

5.1.3

Can you use a single semaphore as a lock? More precisely, can every use-case of a lock be implemented by using a single semaphore? Why?

While it is genereally possible to implement a lock by using a single (binary) semaphore, it may not be the best idea, as mutexes for example serve other purposes as well as provide additional functionality.

Can you directly use a single lock as a semaphore? More precisely, can every use-case of a semaphore be implemented by using a single lock? Why?

Generally, this is not possible. There may be use-cases of binary semiphores that are implemented by using a lock, but as soon as there is more than one resource managed by a (counting) semaphore, it will not be possible to efficiently implement it using a single lock. The main problem in this scenario is, that a single exclusive lock will only be able to manage one resource, so either all the other resources remain unmanaged, leading to a whole other world of problems, or all other resources remain unused, which is horrible from a performance point of view.

Does this mean they are equivalent or is one of them the more powerful abstraction?

There are different use-cases, but generally, speaking, what can be done with locks might also be done with semaphores, and more (refer to ring buffers). As semaphores were introduced by Dijkstra in 1968 to solve problems that could not have

been solved using locks (e. g. ”busy wait”), it is safe to say, that semaphores indeed are the more powerful abstraction.

5.1.4

What is RPC?

RPC stands for Remote Procedure Call and is a concept implemented in networks by communication systems. What it does is calling a function for a return value that is needed by a local machine (client) and executing this function on a server.

What is the idea of RPC?

Procedure calls are calls that transcend address space, meaning a procedure call that is not executed on the machine it was called, but rather on another machine within the network. A communication system is tasked with managing these RPC between client and server, just as it does for IPC. The client calls a function and creates a client stub locally that sends the call over to the communication system as a message. The communication system then figures out the proper host (in case there are more than one possibilities) and passes the message on to the skeleton on the server. The skeleton unpacks the call, the server processes it and returns the return value back to the skeleton, that then packs it within a message. The communication system sends the message over to the original client host, the stub unpacks the message and the return value of the RPC is returned to the client. This takes some procedural load off of the client machine, but heightens the load on the server.

What is a stub in the context of RPC?

Stub in the context of RPC means a representation of the server on the client side and basically is an abstraction of the IPC mechanism. A subprogram on a local machine is substituted by a stub having as identical identifier as the server-sided function has. The stub packs the procedure call into a message and passes that message over to the communication system, so that the call itself is implemented by a message exchange.

What is a skeleton in the context of RPC?

On the server-side the client is represented by a so-called skeleton. The skeleton calls the implemented function on the server-side, unpacks the parameters of the call and packs the return value, which is later unpacked by the client-side stub, after the call’s return was sent via message exchange.