Operating System Concepts Weekly

Week 2

Lucas van der Laan
s1047485

1

A) 0

B) 2603

C) 2603

D) 2600

2

Synchronous and asynchronous communication

Synchronous is good for when you simply want operations to happen 1 after the other, but is very limiting because it is really slow. Asynchronous on the other hand is very good for when processes should communicate quicker and the result is not always necessary, the problem with asynchronous is that developers can more easily make mistakes though.

Automatic and explicit buffering

Automatic buffering makes it easier for the developer to not have to handle the buffer management, this also makes it so that the buffer will never be full thus the sender never blocks, it does however potentially (depending on the implementation) waste a lot of memory, as it will always reserve more memory than it needs, which can either be a lot or not that much.
Explicit buffering however does not have that memory waste, as the developer has defined a set capacity. This has the drawback that the buffer may get full, which means the sender has to block, which leads to waiting time for the sender.

Send by copy and send by reference

Send by copy requires more memory than send by reference, since the copy has to be saved on the stack.
Sometimes we just want to send a deep copy instead of a reference, for example when do make a getter for a list. In this case, we don't want the accessor to be capable of modifying the list, so we send a copy instead of a reference (which by its nature is editable). The opposite is also true, we often want to send the reference (pointer) to another function so that they can work with it directly.

Fixed-sized and variable-sized messages

Fixed-size messages are more straightforward for the system, but are more difficult to program. Variable-sized messages require a more complex implementation for the system, but are easier to program for the developer.

3

The benefits are twofold:

- Using processes for different tabs makes it so that if a tab crashes, only the renderer of said tab crashes and the rest doesn't.
  There will be no difference between a thread and a process in this case, because you can handle a thread in the same manner in which you discard it if it crashes.
- Every process (tab) runs in a sandbox so access to (network) IO and the disk are restricted. Here the process shines and this only holds if we use processes instead of threads, since processes can run in isolation while threads can not. Threads run on the isolation provided by their process.

4

- Yes, because a single processor can still handle multiple threads, so we can do other things while we are waiting for disk I/O. Best case scenario, this can change the turnaround time from 10 seconds to 4 seconds + the time it takes to handle the I/O.
- The only difference between 1 and 4 cores is that the computer with 4 cores will have to wait less time in the ready Queue before it gets allocated back to a processor.

5

There are three paths in this program, as we fork twice.

- Parent: i = 0; i = i + 1; i = i + 1; $\rightarrow$ i = 2
- Child1: i = 0; i = i + 1; fork(i = i + 1); i = i + 1; $\rightarrow$ i = 3
- Child2: i = 0; i = i + 1; fork(i = i + 1; fork(i = i + 1)); i = i + 1; $\rightarrow$ i = 4

Thus it prints out in order:
i is 2
i is 3
i is 4

- The parent never goes inside a fork statement, but it will continue to run the code.
- The child will run the entire code again and will remember that in line 4 it had pid == 0, as it is the child.
- The next child will do the same but remember in line 6 instead.