Joachim Grimen Westgaard

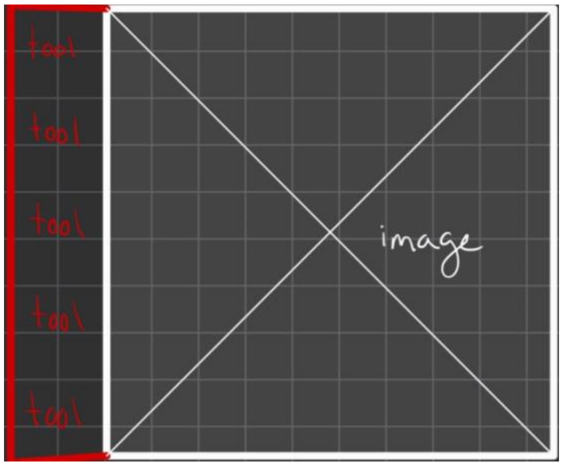
Assignment 2

1 Processes and threads

1. Threads are individual/independent sequences of instructions running within a program. There can be multiple threads set up per process, for example within a process which is set to complete multiple different tasks at the same time (fetching data from a database while also drawing on the screen like Google Maps).
   The thread itself is simply instructions that the kernel executes in user mode, while it is the process itself that runs the system calls to the kernel to access privileged operations on behalf of the process itself. A thread itself is not a process, because it does not have the same privileges that a process has, while also only being a *part* of a process.

   The kernel itself can also benefit from the use of threads, where every kernel-thread will run its course having kernel-level privileges.

2. Consider the following: we want to run a program that fetches an image and adds tools for manipulation of the image in a sidebar next to the image (see image below for reference).



In this program, we might want to speed up the loading process of the GUI, so we split the program's work into multiple jobs. One job is to fetch the image and then draw it on the screen. Another job can be to draw the tools and sidebar next to the image. We can see that the second job of drawing the tools on the GUI does not need the image to be fetched and displayed before execution. This means that we can split the program into multiple threads or processes to speed up the loading of the program. If we were to split the program's process into multiple threads, the threads will all use the same memory area of the program's single process but have different instruction sequences. One thread can start with the drawing of the tools on the sidebar, while another thread waits for the image to be fetched from e.g. a database. A simple program like the one described above, might be best suited for parallel processing by splitting the process into multiple *threads*.

Splitting a program's process is often faster than splitting the process into multiple processes, as the threads will all occupy the same memory region within the single process, while the multiple processes will scatter themselves over multiple memory addresses.

We might want to use multiple processes for multi-processing when dealing with processes that should not necessarily have access to each other's memory spaces. In addition to this, threads are often limited to usage on a single machine, while processes can be split up and run on different machines at the same time.

Processes generally generate more overhead, where threads create less overhead. Context switching between processes is also slower than those of threads, which might lead to the idle time between a processes instructions end up being unseized.

3. When a thread is running, there might be some periods where the CPU core currently in use by the thread might want to give its attention to another thread, for example due to priority. When the core switches between threads, it needs a way to ensure that the threads' states are saved in between working with them. This is where the TCB (Thread Control Block) comes in, which will save the thread's current state before the core shifts its focus elsewhere. When it returns its attention to the original thread, it can quickly continue from where it left off, by reading from the TCB's "current state" data.

4. *Cooperative threading* refers to a system's thread scheduler being centered towards allowing each thread to complete its sequences of instructions without interruptions from other threads. This essentially means that the CPU it is running on won't be available for other threads before the thread has finished its instructions. No other threads will be able to run or affect the

system. This facilitates a sort of control over the threads. However, it has disadvantages. Threads with a lot of instructions can monopolize the CPU for an extended period, which might cause delays seen from a user perspective. A context switch in this thread system can only happen when the current thread has completed its instructions.

The opposite of this type of thread system, can be referred to as *pre-emptive*, where the running threads can be swapped at any time the OS deems fitting. This is what we think of as *multi-threading*. For a context switch to happen in this type of thread system, the OS only needs to find it fitting for another thread to receive access to the CPU. As mentioned earlier, this can be due to for example priority or the fact that the current thread needs to wait for data from the disk, where the CPU in the meantime can complete instructions of another thread and then switch back when the original thread has received its data.

2 C program with POSIX threads

1. The function *go()* is the function that is called and ran. This function is ran by each thread individually, and will print the ID of the thread (in this case, the number it is assigned in the first for-loop, a number between 0 and 9). After the thread's go-function is ran and its ID printed to stdout, the function calls for the thread to exit its lifecycle with an ID of its original ID plus 100.

2. Because the choice of which thread gets to occupy the CPU enough for the function to be finished can be different for each run.

3. When thread 8 prints, the threads 0 and 1 have completed their instructions. We started with 11 threads (including the main thread). This means that when two of these 11 threads have completed their instruction sequences, there can exist a *maximum* of 9 threads (this is considering the example output in the task description).

   Technically, the thread 8 can be printed both first and last of all the threads, which means that the *minimum* number of threads that could exist when it prints "hello", is 11 minus all the other threads (excluding the main thread) which results in 2 (thread 8 and the main thread).
   The theoretical *maximum* number of threads will be 11 (counting thread 8 itself as it has yet to run pthread_exit()), if the thread 8 prints its result first.

4. The function *pthread_join()* is used to merge the threads into the original main thread. It will, due to the implementation of the for-loop, merge them in order starting with thread 0. The function itself will wait for the thread to complete its instructions, before then returning the value that was passed to the thread through *pthread_exit()*.

5. The other threads will continue to run their courses, but the *pthread_join()* for thread 5 will have to wait for the thread to complete its 2 seconds of sleep before joining into the main thread. This will result in there being a slight delay before the final print statement ("Main thread done.") prints, as the program needs to wait for thread 5 inside the join function in the for-loop.

6. When the *phtread_join()*-function returns, the thread will go from the "running" state to the "finished" state, and will transfer its *pthread_exit()*-value to the long *exitValue*. In other words, the thread X has finished running and will exit when the join-function returns.