Assignment 2

1 Processes and threads:

1. **Explain the difference between a process and a thread:**
    a. A process is a program in execution, while threads are segments of processes. Threads are more lightweight as they share memory and can therefore also use shared memory instead of IPC for communication which reduces the number of system calls.
2. **Describe a scenario where it is desirable to write a program that uses:**
    a. **Threads for parallel processing:**
    When you have lightweight tasks in terms of memory size and processing time, for example a web server handling page requests. It's also useful when different parts of the same information are processed as they can share the same memory.
    b. **Processes for parallel processing:**
    When the tasks/workloads use significant memory and/or processing power. Also, when handling different info that shouldn't affect each other or share memory, like different tabs in a browser, multiple processes can be desirable.
3. **Explain why each thread requires a thread control block (TCB):**
    a. The OS needs a data structure to represent a thread's state. The TCB contains the state of the computation being performed and metadata about the thread. This is so the OS is capable of being able to start, stop and managing each thread as needed.
4. **What is the difference between cooperative threading and pre-emptive threading?**
    a. With cooperative threading, each thread manages its own lifecycle. Which means that its up to the thread itself when to stop afters its been started. With pre-emptive threading the scheduler has more control over each thread so the threads can be stopped by the scheduler when it yields or has used its given time slice.
    Since threads can get interrupted be the scheduler before it's finished with its task, we have to store and restore the threads state when it is interrupted so it can continue at the same point later. This is a context switch and it takes more time in pre-emptive threading than in cooperative threading, because threads in cooperative threading manages their own lifecycle and the scheduler therefore doesn't have to keep track of the thread state. Hence, the context switches are much faster.

2 C program with POSIX threads

1. **Which part of the code (e.g., the task) is executed when a thread runs?**
    a. The function *go* is runned. *go* prints "Hello" and which thread its running from, and then exits with a return value of which thread it's in.
2. **Why does the order of the "Hello from thread X" messages change each time you run the program?**
    a. Because the scheduler is pre-emptive and what order the threads should run in therefore aren't necessarily the same as the order the threads was created in.
3. **What is the minimum and maximum number of threads that could exist when thread 8 prints "Hello"?**

   a. The main thread creates 10 threads. We don't know which order they are run in, so there can be maximum 1 (main) + 10 (all created) = **11** threads existing and minimum 1 (main) + 1 (thread 8) = **2** threads existing.

4. **Explain the use of *pthread_join* function call:**
   a. The *pthread_join* function suspends execution until the given thread terminates and if the *pthread_exit* contains a non-NULL *value_ptr* argument, it's made available to *pthread_join*. With *pthread_join* we can get notified when a thread ends and e.g. print it like in this case.
5. **What would happen if the function *go* is changed to behave like this?**
   a. When thread number 5 is executing it will pause the thread for 2 seconds, which means that when printing "returned with" it will pause for 2 seconds between thread 4 and 5 because the program checks for completion in the order that the threads were created. The "Hello from" prints can still be in random order.
6. **When *pthread_join* returns for thread *X*, in what state is thread *X*?**
   a. Thread **X** is in *FINISHED* state when *pthread_join* return for it.