# Operating systems – Task 2

Threads & Processes

Ingebrigt Hovind

## 1 Processes and threads

1. A process is made up of one or more threads. A thread is a single execution sequence executed sequentially. A process is a program in execution with its own PCB, which is where the information that the operating system needs about the process is stored. A thread is a segment of this process, inheriting parts of its PCB, such as its heap, while creating its own TCB, containing its own stack, program counter and thread id.

   Threads are lightweight processes which require less overhead in order to create compared to forking a new process. A Thread has access to its parent's memory, but it gets its own stack, while a new process copies on write if it references parts of memory belonging to the process that created it

2.

Threads are advantageous when wanting to do more than one thing at a time in a program. For example, a word processor might want to save the document you are writing in the background without freezing the program for the user. The developer can then use separate threads, one to handle the program itself, and another one which is created at given intervals in order to automatically save the document to disk.

A thread would be advantageous if the developer wants to do more than one thing at once, but does not want them to have access to each other's memory or to interact with each other an excessive amount. While processes use more resources to create and run, they are more secure in the way that they do not share their memory like threads do. If a programmer want to ensure that a bug in one sequence of instructions does not affect another part of the program, then the programmer needs to create a separate thread.

3. Each thread requires a thread control block in order to keep track of state of the execution of the thread. Without keeping track of this, it would be impossible for the kernel to start a thread after stopping it, as there would be no way to know where to restart it from. A thread also requires its own stack, as a thread is a separate execution from the parent process with potentially its own method calls and local variables.

4. Cooperative threading means that the thread itself decides when to yield the processor and allow it to work on other threads, this allows the thread to potentially hog the processor, slowing the system down significantly.

   Preemptive threading means that the kernel decides when to yield a thread, so that no single thread can keep the processor busy for an inordinate amount of time.

   In each case the context switch requires kernel privileges. Firstly the state of the thread is saved to memory, then the kernel's handler code is run in order to handle the interrupt or exception that caused the context switch. Lastly the thread that is to be run next is restored onto the processor, so that it may resume running where it left off.

## 2 C program with POSIX threads

1. The go function is executed when the code runs, as the thread is created with a pointer to this function as a parameter. What the go function does is print the hello from the thread including the thread number, and then exit with the exit code 100 + (thread number)
2. Because creating and scheduling threads are two separate processes. This means that threads do not have to be scheduled in the order that they are created, as the kernel can force the thread to yield at any moment in order to prioritize another thread.
3. The maximum number of threads that could be active when thread 8 prints "hello" would be 11, as the program creates ten threads, which in addition to the original thread of the program makes 11. Due to the scheduling of the threads not following a set order, the program could create all the threads, then thread  could print "hello" before an of the threads have been closed. Following the same logic, the minimum number of threads when thread 8 prints "hello" is two, as thread 8 could be scheduled in a way that every other thread is closed before it has a chance to print hello, leaving only it and the original thread.
4. pthread_join halts the main thread until the given thread has had time to complete. Using this in a for loop as shown ensures that all the threads exit in order of creation
5. This would make thread 5 sleep and halt it's execution for 2 seconds. As this happens after the printing, the greetings would still happen in an arbitrary order. But the exits would be ordered just like in the original program. Sleeping like this would practically speaking ensure that all the threads perform their greetings before thread 5 has exited, as even if thread 5 prints before any of the other threads, the other threads would then be scheduled while the processor awaits thread 5 to be done sleeping.
6. When pthread join is executed the threads are either currently running or finished. If the thread is not finished then the main thread is halted until the given thread finishes. So when the pthread_join returns, then the thread it took as a parameter is in the finished state