1 Processes and threads

Text

Description automatically generated with medium confidence

A process is an instance of a program and a process can have multiple threads running at the same time. A process is started and managed by the operating system while the program can start and manage threads (the user or the kernel)

Graphical user interface, text, application

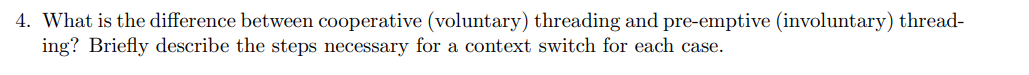
Description automatically generated

This could be helpful when having simultaneous tasks running, two while loops, responsive gui (GUI can be responsive while other tasks are running in the background), servers can serve multiple requests).



Modern web browsers do this for each tab/window for security reasons

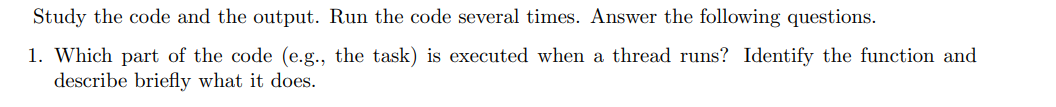


To make it easier for the operating system to manage the threads. A thread control block consists of information about the thread like thread id, its priority, status, program counter etc. and these can be used by the operating system to properly manage the threads and schedule them as desired by the OS

Cooperative/voluntary threading requires yielding the CPU explicitly, for this to work all programs must cooperate. The thread can then run again later when it gets the chance to do so.

Pre-emptive/involuntary threading requires yielding the CPU because of some other factor, like timer interrupts or system calls. When this happens, an interrupt handler comes in and ultimately chooses the next thread in queue.

Graphical user interface, text, application, email

Description automatically generated

Under the purple line is the creation of the threads with the use of pthread. The function go is the one being called within the threads for each thread, and it prints out the thread number (the one we assigned to it).

Text, table

Description automatically generated with medium confidence



Even though creation of threads is sequential (in written code) it Is not guaranteed they run/finish in the order we gave them. Here the threads are initialized in order but after creation, the scheduler could have been stalled so that they couldn’t print in the order we wanted.



In total we have 11 threads, 10 of them we create and 1 is default for the starting main thread (to be able to start and run the program).

Max is therefore 11, including thread 8’s thread (because we are not done with 8 when printing).

Minimum is 2, which includes thread 8 and the main thread.



We see that the join function is in a for loop under the creation for loop. This is to make sure that we don’t end the main function too early and exit the programs before the threads have completed their task, and to do that we need to wait for the threads to finish. We do that using pthread\_join, and once every thread is done with its task, the main function prints “Main thread done” and exits the program with a return. Important to note that we check and wait for the threads sequentially, from thread 0 to thread 9, which we can confirm by the order of “thread x returned with 100+x” (they are always in order).

Text

Description automatically generated

The creation of thread 5 and above won’t be halted since the wait of 2 seconds would only affect thread 5 in its own thread. But when using the join function, the way it is already implemented in the program here, would halt thread 6-10. The join function waits on the threads to finish one by one before going to the next and therefore threads 6-10 would get 2 seconds delay before they can be joined.



Thread X would be in a finished state, and this would be saved in the Thread control block with a status of done or finished. This means that the thread has done its task and should not occupy/run in its own thread anymore.