# IDATT2202 OBL2-OS

Nicolai H. Brand

October 2022

## 1

### 1.1

A process is a running instance of a program. A thread is a single execution sequence withing a running process. In other words; a process has at a minimum one or more threads that execute instructions.

### 1.2

When creating a program with some sort of GUI it is essential to gather mouse input from the user. To ensure responsiveness, the program can run two threads. One main thread and a secondary thread whose responsibility is to gather mouse input from the user.

When a program can be divided into multiple completely isolated and independent instances it may be natural to do parallel programming using multiple processes. For example, a webapp usually has a distinct backend and frontend that run on separate processes.

### 1.3

The thread control block (TCB) stores the state of the thread. This information is necessary for the OS to be able to correctly start running the thread, or continue running a thread that was pre-empted. For example, if a thread was stopped before it was finished, the registers in the CPU needs to be saved so they can be restored once the thread continues its execution. In addition, the TCB stores the running state of the thread - ready/waiting/running/finished.

### 1.4

Involuntary threading is the more programmatic way of implementing treads. The execution order, and for how long, is entirely handled by the operative system kernel. Therefore, whenever the operative system kernel decides to switch context it sends an interrupt to the thread which is handled by the interrupt handler.

Cooperative threads decide for themselves how long they are being executed. The responsibility of how long the thread is running for is therefore up to the thread itself. For a context switch to occur, the thread has to voluntarily yield. A properly behaving cooperative thread will yield periodically to account for this, however a badly implemented cooperative thread can simply ignore this.

# 2

## 2.1

Every thread executes the function `go`. Inside the function, the thread simply prints its assigned index $n$ and exits with the exit code $100 + n$.

## 2.2

The order of what thread runs before another is not necessarily determined by the order of when they were created. In addition, a thread may be pre-empted before its execution sequence is finished.

## 2.3

The minimum is 2. Thread 8 may be the first to be executed and therefore no other threads will have executed other than thread 8 itself and the main thread. The maximum is 11. This is because all threads, including the main thread, may exist once threa 8 prints.

## 2.4

The `pthread_join` function call waits for the thread specificed to be terminated. In this case, it is being used to to wait for threads to finish, in order. This is why thread 2 cannot return before thread 1.

## 2.5

Whenever thread 5 starts executing, it will be put to sleep for 2 seconds. While thread 5 is sleeping, the operating system kernel is free to pre-empt it and start executing another thread. However, thread 5 is guaranteed to sleep for 2 seconds, and no other thread after thread 5 can return before the sleep is finished.

## 2.6

After calling `pthread_exit`, the thread is in its *finished* state.