# OBL2 - Operating Systems

**By:** John Ivar Eriksen

## 1 - Processes and threads

### 1.1

> Explain the difference between a process and a thread.

The difference between a process and a thread (p.137) is that a process is the execution of an instance of a program, with restricted privileges. while a thread is a segment of a process, an independent sequence of instructions, running within the program in question.

### 1.2

> Describe a scenario where it is desirable to: Write a program that uses threads for parallel processing.

If making a program that does "many things" at the same time. E.g. a word processor, where you want to draw and update the GUI at the same time as the document is being saved automatically, and that the spelling checker is running. If all of these had to run in sequence, the use of the program would be very slow and unresponsive, since you'd have to wait for the spellcheck to complete before a menu you clicked would be opened (i.e. drawn on screen).

The example from the book (p.133) is of an Earth Vizualiser application, i.e. an interactive map where the user can navigate around a model of the earth, made from sattelite imagery. Threads will let the program draw the map at different resolutions and areas in the background, wich gives the user the impression of "flying" along when navigating around and zooming in/out. If each action had to be exectued sequentially, it would be very stuttery and unresponsive to navigate.

> Describe a scenario where it is desirable to: Write a program that uses processes for parallel processing.

When you need program where you want parallel processing to be isolated, in that each "sub-process" is isolated from the other sub-processes. Unlike with threads, processes does not share data and information, and have it's own assigned memory area, stack and data.

A scenario where this is desirable, coulb be web servers and databases. Here we would want to separate the work to handle multiple clients, without having to run all the clients under one process, i.e. with threads. Instead, if they are separated into their own processes, any errors or mishaps in one process will (most likely) not affect the other clients' processes, since they are isolated. This would also be the case for online game servers, which would want to isolate sessions/clients from each other.

## 1.3

> Explain why each thread requires a thread control block (TCB).

Each thread required a TCB in order for the operating system to keep track of the state of any given thread (p.146). The TCB holds information about 1) the state of the computation being performed by the thread, and 2) metadata about the thread which is used to manage it.

## 1.4

> What is the difference between cooperative (voluntary) threading and pre-emptive (involuntary) thread-
> ing? Briefly describe the steps necessary for a context switch for each case.

Cooperative threading is when a thread runs without interruption until it "voluntarily" give up control of the process to another thread (p.139). This means that in most cases, for cooperative multi-thread systems, only one thread will will run at a time, and no other threads will be able to run.
Pre-emptive threading is when threads can be switched at any time, without the process/thread itself needing to "consent" to it, i.e. it must involuntary give up control to another thread.

# 2 - C program with POSIX threads

## 2.1

> Which part of the code (e.g., the task) is executed when a thread runs? Identify the function and
> describe briefly what it does.

When a thread runs, the `go` function is executed. This function is designated as the `start_routine` for each thread created in the `main()` function.

- The asterisk `*` is used to declare a pointer to `void`, i.e. a generic pointer of no specific data type.
- The `go` method is declared to return a pointer to `void`, with `void *`.
- `go` takes in the argument `void *n`, where `n` is the iterator value `i` being passed from `main()`.
- `i` is cast to a generic pointer with `(void*)i` in order to be passable to `go`
- `n` is used to identify the tread, and is cast to a `long` in the print statement

## 2.2

> Why does the order of the “Hello from thread X” messages change each time you run the program?

The order of the messages change every time the code is run due to concurrence. The threads and scheduling are not executed in a specific order, and each thread will execute at different speeds and priorities depending on "external" factors that influence the scheduling descisions, like resource availability.

## 2.3

> What is the minimum and maximum number of threads that could exist when thread 8 prints “Hello”?

Minimum number of threads are 2: The main thread (parent), and thread 8 itself (child thread).
Maximum number of threads are 11: Since threads are not executed in any specific order, all other threads 10 may be existing when thread 8 prints "hello". Total threads are the main thread, plus child threads 0 through 9.

## 2.4

> Explain the use of `pthread_join` function call.

**Synchronization:** `pthread_join` function in POSIX threads (pthreads) acts as a synchronization mechanism which allows a thread to wait for the termination of another thread, ensuring an orderly execution of threads in concurrent programming where the execution of one thread depends on the completion of another thread
**Exit status:** It also allows retrival of the exit status of the terminated thread, which allows the terminated thread to communicate data back to another thread, which ties into the synchronization mentioned above.
**Resouce reclamation:** `pthread_join` ensures that resouces from terminated threads are reclaimed by the system, which ensures that we use system resources properly and don't run into memory leaks from e.g. zombie threads or the memory space occupied by the terminated thread not being reclaimed and reused.

## 2.5

> What would happen if the function go is changed to behave like this:
> [modified code block]

Compiler output of the modified code:

```
§ gcc -o threadHello2 threadHello2.c -lpthread

threadHello2.c: In function ‘go’:
threadHello2.c:9:10: warning: comparison between pointer and integer
    9 |     if(n == 5)
      |          ^~
threadHello2.c:10:5: warning: implicit declaration of function ‘sleep’ [-Wimplicit-function-declaration]
   10 |     sleep(2); // Pause thread 5 execution for 2 seconds
      |     ^~~~~
```

With the modified code, the program will run until the `if` statements encounters `thread n == 5`. When this evaluation is done, and `if n == 5` is found to be true, the program will pause for 2 seconds, `sleep(2)`, before continuing. The other child threads will continue to execute as normal.

There will, however, be a delay in joining the main thread, and as the `pthread_join` will join threads back into the main thread in sequence, it will pause at thread `n == 5` for two seconds before it proceeds to join the remaining threads (6 through 9)

## 2.6

> When `pthread_join` returns for thread X, in what state is thread X?

When `pthread_join` returns for any given thread, the state of that thread is necessarily `terminated`, as it will have finished its work. The resouces it was using can then be reallocated, and it will return an exit signal.