# OBL2-OS – Kristoffer Longva Eriksen

This is a mandatory assignment. Use resources from the course to answer the following questions. Take care to follow the numbering structure of the assignment in your submission. Some questions may require a little bit of web searching. Some questions require you to have access to a Linux machine, for example running natively or virtually on your own PC, or by connecting to gremlin.stud.iie.ntnu.no over SSH (Secure Shell). Working in groups is permitted, but submissions must be individual.

## 1 Processes and threads

1. Explain the difference between a process and a thread.

A process is an instance of a program, while a thread is a segment of a process. A process is controlled by a PCB while threads use a TCB, and a thread can only be either running, waiting or blocked while a process can also new, ready or terminated. The OS manages processes, while threads are managed by the user or a program, and each program can have multiple threads.

These threads share the programs data, while a process is isolated and doesn't share its data with other processes. This makes the processes threads dependent on each other, unlike processes that only depend on themselves, with its own address space. What threads lack in flexibility they make up for in manageability. They are very lightweight and take less time to create, communicate and terminate than a process.

2. Describe a scenario where it is desirable to:

- Write a program that uses threads for parallel processing

  This is ideal in terms of not putting too much load on the CPU, when there is no requirement for memory isolation. A common example would be a program with input and a GUI, where one thread handles input while another updates the GUI and another performs spellcheck.

- Write a program that uses processes for parallel processing

  If process isolation is required, for example if multiple users are using the same program, requiring sensitive data, processes should be used. This provides security and reliability (we also make sure processes don't write over each other's memory).

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

Thread Control Blocks represent threads in much of the same way a PCB represents a process. It contains a thread ID and the thread state, as well as any other information the OS would need to know how and when to execute it. It also contains information about the threads process and any new threads created in the thread itself.

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

Context switches back to the threads parent or process can be done pre-emptively with an interrupt or exception. This can be required if a thread unknown to the current thread has a higher priority and needs to run, but it is also best to not trust threads to return on their own in case they hold malicious

code and would like to hold up the CPU forever. The alternative to pre-emptive threading is cooperative where we expect the thread to return with thread_yield() or _join(). This would be ideal if all threads could be trusted and had information about all other threads, however, this is not realistic.

When yield is called the current threads registers are saved onto the TCB or stack (this allows it to be run again later). Then the scheduler selects which thread to run next, restores that threads registers and returns to execution. With pre-emptive threading the context switch is similar, however, it starts with an interrupt or a trap. The kernel uses periodic timer interrupts to keep control of the process. First, the interrupt or trap is handled, then the scheduler starts the next thread in the same way as with cooperative threading.

## 2 C program with POSIX threads

*nix operating systems use POSIX threads, which are provided by the pthread library. Consider the following adapted code from the textbook (the code has been modified slightly to use pthread, while the book assumes its own thread implementation).

```
#include <stdio.h>
#include <pthread.h>
#define NTHREADS 10
pthread_t
threads[NTHREADS]; void
*go (void *n) {
    printf("Hello from thread %ld\n", (long)n);
    pthread_exit(100 + n);

    // REACHED?

}


int main() {

    long i;

    for (i = 0; i < NTHREADS; i++)
      pthread_create(&threads[i], NULL, go, (void*)i);
    for (i = 0; i < NTHREADS; i++) {

        long exitValue;

        pthread_join(threads[i], (void*)&exitValue);

        printf("Thread %ld returned with %ld\n", i, exitValue);

    }

    printf("Main thread done.\n");
    return 0;

}
```

We can compile the code and tell the compiler to link the pthread library:

```
$ gcc -o threadHello threadHello.c -lpthread
```

At the command prompt, run the program using ./threadHello. The program gives output similar to the following:

```
Hello from thread 0
Hello from thread 3
Hello from thread 5
Hello from thread 1
Hello from thread 4
Thread 0 returned with 100
Thread 1 returned with 101
Hello from thread 9
Hello from thread 8
Hello from thread 2
Hello from thread 7
Hello from thread 6
Thread 2 returned with 102
Thread 3 returned with 103
Thread 4 returned with 104
Thread 5 returned with 105
Thread 6 returned with 106
Thread 7 returned with 107
Thread 8 returned with 108
Thread 9 returned with 109
Main thread done.
```

Study the code and the output. Run the code several times. Answer the following questions.

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 function *go* is executed. go prints “Hello from thread n”, where n is the argument given in pthread_create() in *main.* It then exits with return value 100+n.

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

Thread creation and scheduling are two different and separate operations. From the perspective of the programmer the only guarantee is that each thread runs separately, however because of CPU load or preemption, speed can vary. This means that even though a thread is created before another it may take longer time to finish and return the Hello message.

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

As explained above, we cannot predict in which order the threads will run, therefore if thread 8 is the last to run, after all other threads have exited, there will be two threads running when it prints hello (main thread and thread 8), and if thread 8 is running while all other threads have been created, but not yet exited, there will be 11 threads running.

4. Explain the use of pthread join function call.

Pthread_join() is a function in the parent thread, that waits for a specified child thread to exit and then returns. In this main() function the threads are looped through from thread 0 through 9, meaning that the pthread_join() will first wait for thread 0, then store its exit value and print, and then move on to waiting for thread 1. This is why the Thread returned with message is in sequential order.

5. What would happen if the function go is changed to behave like this:

```
void *go (void *n) {
    printf("Hello from thread %ld\n", (long)n);
    if(n == 5)

        sleep(2); // Pause thread 5 execution for 2 seconds

    pthread_exit(100 + n);

    // REACHED?

}
```

This would cause thread 5 to print its hello message and then wait two seconds before exiting. The hello messages are still in the same “random” order, while the thread returned message from thread 0-4 are normal. Thread 5 will take two seconds longer to exits, and because of the behaviour explained in the last task, process 6-9 would have to wait for 5 to join before they could join. The only thing we guarantee in a normal environment is that the Thread returned messages from thread 5-9 will always be the last output.

6. When pthread join returns for thread X, in what state is thread X?

When thread X is done running, it exits, returning to a terminated state, with the exit value stored in the TCB and pthread_join() executed.