Lab 7
Microprocessor Architectures [ELEC-H-473]
Multithreading, Multiple Processes & Security

v1.0.2

For Apple Silicon based CPUs

The *pthread* library is only available for x86 cpu architectures. Students that have an Apple Silicon based CPUs (ARM based), should still be capable of doing the lab using the Rosetta 2 translation layer available in macOS.

To compile and run your code through Rosetta, open a new terminal and type the follow command:

```
arch -x86_64 /usr/bin/env bash
```

This code will open an x86 terminal allowing you to compile and run code for that architecture. This should allow you to run your code "seemlessly" on your Mac. Note that you will need to do this for each terminal instance, and that this method may impact the performance metrics.

Description of the lab

Objectives:

- Part 1 (Multithread):
  - Learn how to create multiple threads.
  - Quantify potential acceleration.
  - Understand the limits of acceleration due to memory bottlenecks.
- Part 2 (Multiple Process & Security):
  - Learn how to create multiple processes.
  - Quantify acceleration of process-based parallelism in comparison with multithreading.
  - Understand the limits of acceleration due inter-process communication.
  - Understand the benefits of process-based parallelism for isolation and security.

Exercise

Multithreading

1. Read and execute the source code "multithread.c", provided on the UV, and understand different concepts implemented.
2. Understand the thread manipulation functions (thread creation, mutex).
3. Start from the exercises that you have already completed in the previous session (threshold function).
4. Initiate 2 and 4 threads that work on different image subsets, and measure the acceleration over linear processing.

Multiple Process & Security

1. Read and execute the source code "multifork.c", provided on the UV, and understand different concepts implemented. The example is strongly based on Part 1 but uses processes as an alternative mechanism to implement parallelism.
2. Understand the `fork()` and `waitid()` system calls.
3. Observe and explain the output variable states in `myThreadFun()`.
4. `myThreadFun()` contains behaviour that creates randomized crashes of worker processes (l. 39 ff.). What exactly happens when a worker process crashes? How does that differ from the behavior of a crashing thread? Do you see applications of the observed behaviour for system resilience

or cybersecurity? Modify the code that waits for termination of cild processes (l. 76 ff.) to output statistics regarding the number of successful vs. crashed children via `waitid()`.

5. Starting from your results in parallelising the threshold function with threads in Part 1, try and reproduce the behaviour with processes. Fork 2 and 4 processes that work on different image subsets and compare performance and complexity of the implementation with your results from Part 1.

1 Assignment

For this lab, we ask you to submit:

1. Your source code
2. An evaluation and discussion of the different implementations (original vs. multihreaded vs. multi-process)
3. A discussion on multi-threaded & multi-process security aspects.