UNIVERSIDADE DE COIMBRA
UNIVERSITY OF COIMBRA

DEI
DEPARTMENT OF INFORMATICS ENGINEERING

# Operating Systems: Project #3

## Introduction

This assignment will focus in the following topics:

- Concurrent programming with threads, using the `<pthread.h>` library.
- Use synchronization primitives such as semaphores and condition variables

## Objectives

At the end of this project, students should be able to:

- Create and manage multiple threads inside a process.
- Use the library `<semaphore.h>` to create and use semaphores, *mutex* and condition variables to synchronize the access to shared resources between competing threads.

## Support Material

- K. A. Robbins, S. Robbins, "Unix Systems Programming: Communication, Concurrency, and Threads", Prentice Hall:
    - Chapter 12 – "POSIX Threads"
    - Chapter 13 – "Thread Synchronization"
    - Chapter 14 – "Critical Sections"
- "Programming in C and Unix", available at http://gd.tuwien.ac.at/languages/c/programming-dmarshall/:
    - Threads: Basic Theory and Libraries
    - Processes and Threads
    - The POSIX Threads Library: `libpthread`, `<pthread.h>`
    - Further Threads Programming: Synchronization
    - Threads and Semaphores
- Online Unix manual
    - Man pages of `sem_post, sem_wait, sem_open, sem_init.`
    - Man pages of `pthread_create, pthread_exit, pthread_sigmask, sigwait.`
- Course materials:
    - Notes on Unix basic commands
    - Notes on reading and writing to files
    - Notes on `Makefile`

## Project Assignment

### Overview

The objective of this assignment is to rewrite the program you have just developed for project 2 using a **multi-threaded** approach and **new synchronization primitives**, such as Condition Variables. You must also conduct a performance evaluation of this program and assess the speedup attained with the concurrent version. For measuring **speedup**, we supply a sequential version of the program without the snapshots system. Since this work is thread-based, you do not have to use share memory. But, you still have to produce snapshots of execution as required in project 2, thus the use of Memory Mapped Files is recommended.

In this project you will implement the Game of Life (http://en.wikipedia.org/wiki/Conway's_Game_of_Life) by distributing the work of computing each generation by N threads. But, you must **avoid using a barrier-like** synchronization approach of **all threads** when moving from a generation to the next. Consider Figure 1.

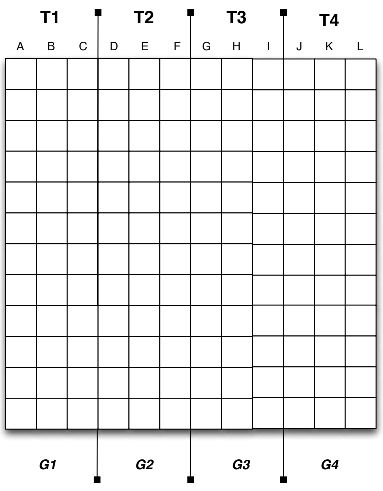

Figure 1 – Distribution of work

Figure 1 represents a matrix 12x12 cells. T1, T2, T3 and T4 are threads in charge of computing the new generations of values for the cells in 4 distinct sections of the matrix (you can use a different number of threads/sections in your program). G1, G2, G3 and G4 are variables that hold the identification of the last generation completed for each section of the matrix individually. Synchronization must follow these rules:

1. A thread can start calculating the value of its inner cells for a new generation as soon as it completes the previous one;
2. A thread can start calculating the value of its cells on the right, near the section frontier, as soon as the thread on the right finishes the previous generation;
3. A thread can start calculating the value of its cells on the left, near the section frontier, as soon as the thread on the left finishes the previous generation;
4. When doing a snapshot, all sections of the matrix need to be in the same generation.

Let us consider some examples:

1. T1 can start calculating generation N of the cells in column B immediately after it finishes calculating generation N-1 (G1 = N-1);
2. T1 can start calculating generation N of the cells in column A immediately after T4 finishes calculating generation N-1 (G4 = N-1);
3. T1 can start calculating generation N of the cells in column C immediately after T2 finishes calculating generation N-1 (G2 = N-1);
4. T2 can start calculating generation N of the cells in column D immediately after T1 finishes calculating generation N-1 (G1 = N-1).

You must use Condition Variables for performing the synchronization between threads. Solutions with potentially higher levels of parallelism will receive extra points.

## Descrição em Português

O objectivo deste projeto é reescrever o programa desenvolvido no projeto 2 utilizando uma abordagem *multi-threaded* e novas primitivas de sincronização, com por exemplo, Variáveis de Condição. Também será necessário realizar uma avaliação da performance do programa desenvolvido e verificar os valores de *speedup* alcançados. Para permitir uma análise comparativa do *speedup*, será fornecida uma versão sequencial do programa a desenvolver. Sendo que, esta não incluí o mecanismo de *snapshots* desenvolvido no projeto 2. Como o contexto de execução de um processo *multi-threaded* é baseado em memória global, não será necessário utilizar memoria partilhada neste projeto. No entanto, ainda é necessário desenvolver o mecanismo de *snapshots*, pelo que o uso de *Memory Mapped Files* é recomendado.

Neste projeto irá implementar o Jogo da Vida (http://en.wikipedia.org/wiki/Conway's_Game_of_Life) distribuindo o trabalho de calcular as novas gerações por N *threads*. No entanto, deve evitar um mecanismo de sincronização do tipo barreira que envolva a paragem de todas as *threads* antes da barreira. Para perceber melhor o que é esperado deste trabalho, considere o exemplo na Figura 1.

A Figura 1 representa uma matriz de 12x12 células. T1, T2, T3 e T4 são as *threads* responsáveis por calcular as novas gerações em 4 subáreas da matriz (no seu trabalho pode acrescentar mais *threads* e áreas). G1, G2, G3 e G4 são variáveis que mantêm o registo do número da última geração terminada para cada subárea da matriz. A sincronização deve obedecer às seguintes regras:

1. Uma *thread* pode iniciar o cálculo de uma nova geração das suas células interiores assim que terminar a sua geração anterior;
2. Uma *thread* pode iniciar o cálculo da próxima geração nas suas células junto da fronteira direita assim que a *thread* à direita terminar o calculo da geração anterior;
3. Uma *thread* pode iniciar o cálculo da próxima geração nas suas células junto da fronteira esquerda assim que a *thread* à esquerda terminar o calculo da geração anterior;
4. Quando se realiza um *snapshot*, todas as *threads* devem ter concluído o mesmo número de gerações.

Alguns exemplos:

1. T1 pode iniciar o cálculo da geração N das células na coluna B assim que tiver terminado o cálculo da geração N-1 (G1 = N-1);
2. T1 pode iniciar o cálculo da geração N das células na coluna A assim que T4 tiver terminado o cálculo da geração N-1 (G4 = N-1);
3. T1 pode iniciar o cálculo da geração N das células na coluna C assim que T2 tiver terminado o cálculo da geração N-1 (G2 = N-1);
4. T2 pode iniciar o cálculo da geração N das células na coluna D assim que T1 tiver terminado o cálculo da geração N-1 (G1 = N-1).

Devem utilizar variáveis de condição para realizar os principais passos de sincronização entre *threads*. As soluções que ofereçam um grau de paralelismo potencialmente mais elevado receberão pontos extra.

## Calculating speedup

Speedup is a metric defined by the following formula:

$$S_P = \frac{T_1}{T_P}$$

Where $S_P$ is the speedup obtained when executing the program in a system with $P$ cores or CPUs, $T_1$ is the execution time of the sequential version of the program, a $T_P$ is the execution time of the concurrent program in a machine with $P$ cores or CPUs.

## Quality requirements

Your application must make use of:

(a) Multi-threaded process;

(b) POSIX synchronization primitives;

(c) Solutions using Condition Variables for managing thread progression are mandatory.

The program should also:

(d) Provide the results of the performance analysis and the attained speedups in your report;

(e) Include a snapshot mechanism for validation of functionality.

To fulfil the objectives of this assignment you should be as creative as you want, provided you have included this list of features in your solution.

## Notes

- **Plagiarism or any other kind of fraud will not be tolerated**. Attempts of fraud will result in a ZERO and consequent failure in the OS course.
- Do not start coding right away. Take enough time to think about the problem and to structure your design;
- Implement the necessary code for **error detection and correction**;
- **Avoid busy-waiting** in your code!!
- Assure a **clean shutdown** of your system. Release all the used resources;
- Use a `Makefile` for simplifying the compilation process;
- Be sure to have some debug information that can help us to see what is happening.

Example:
```
#define DEBUG  //remove this line to remove debug messages
(...)
#ifdef DEBUG
  printf("Creating shared memory\n");
#endif
```

## Submission instructions

1. The names and student ids of the elements of the group must be placed at the top of the **source code file(s)**. Include also the time spent in this assignment (sum of the time spent by the two elements of the group).
2. Any external libraries must be provided with the source code file, including the usage instructions. Use a **ZIP** file to archive multiple files.
3. You must provide a short **REPORT** (maximum 2 A4 pages) explaining the design and implementation of your solution and justifying the design options that you have made.
4. The project should be submitted to InforEstudante.
5. The submission deadline is **December 9, 2012 (23h55)**