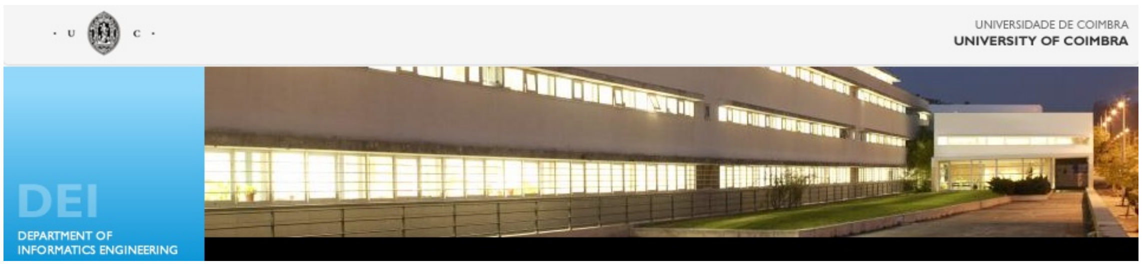


Operating Systems: Project #2

Introduction

This assignment will focus on the following topics:

- Shared Memory as a mean for sharing data between processes;
- Memory Mapped Files, as a way to manipulate files on disk and also share data between collaborative processes;
- POSIX Semaphores, for controlling accesses to shared resources.

Objectives

At the end of this Project students should be able to:

- Create and use a segment of shared memory to share data between processes;
- Map a file into a buffer in memory and use it to share data between processes;
- Use the library <semaphore.h> to create and use semaphores to synchronize the access to shared resources between concurrent processes.

Support Material

- K. A. Robbins, S. Robbins, "Unix Systems Programming: Communication, Concurrency, and Threads", Prentice Hall:
  - *Chapter 14 – "Critical Sections and Semaphores"*
  - *Chapter 15 – "POSIX IPC" (sections 15.1 a 15.3)*
- W. Richard Stevens, Stephen A. Rago, "Advanced Programming in the UNIX® Environment: Second Edition", Addison Wesley, 2005
  - 14.9 – "Memory-Mapped I/O"
- "Programming in C and Unix", available at http://gd.tuwien.ac.at/languages/c/programming-dmarshall/:
  - IPC:Semaphores
    - POSIXSemaphores:<semaphore.h>
  - IPC:Shared Memory
  - IPC:Shared Memory - Mapped memory
- Online Unix manual
  - Manual pages of sem_post, sem_wait, sem_open, sem_init.
  - Manual pages of fork, exec.
  - Manual pages of shmget, shmctl, shmat.
  - Manual pages of mmap, munmap
    • Course materials: o Notes on Unix basic commands;Notes on reading and writing to files

Project Assignment

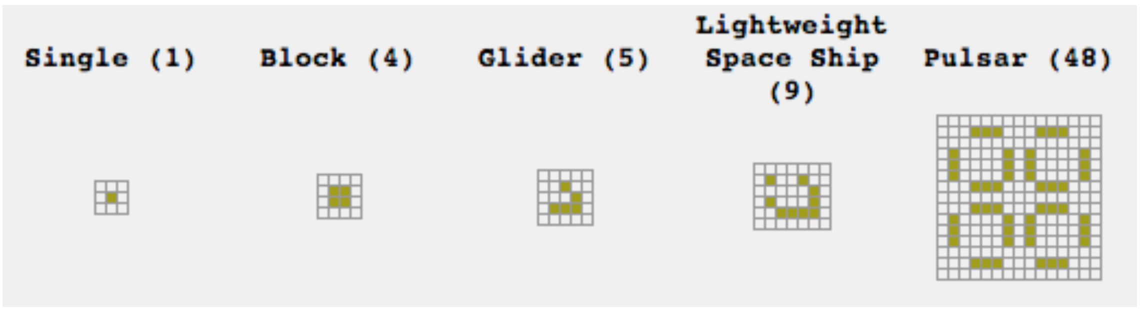
Overview

In this project, you will implement a multiplayer version of Conway's Game of Life (http://en.wikipedia.org/wiki/Conway's_Game_of_Life). Traditionally, Life is a zero-player game because it plays by itself. The game works on a grid and each cell updates based on the following rules and the 8 neighbors cells.

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

A MultiProcess Approach

One of the beauties of the Game of Life is that following the existing rules several patterns emerge. Below are some examples of patterns that either disappear (Sing), stationary (Block and Pulsar) or move through the grid (Glider and LSS).



In your project, you will have a common grid that will be shared by several processes using either **shared memory** or **memory-mapped files**. Each process will be responsible to handle part of the grid and make sure the shapes evolve according to the rules. You should be aware that the updates on the grid by one process must be synchronized if another process is updating neighbor cells at the same time.

Locking Granularity

There are several options when synchronizing accesses to cells. If you synchronize on each cell, you will have too many locks, but a good performance. If you synchronize the whole board, programs will be very slow. You should find a common ground for your project and justify it in the report.

Visualization

In the beginning of the program, the user may supply for how long should be able to define the following game parameters:

- Duration of the Game
- Frequency of Snapshots
- Dimensions of the Grid (you can consider the grid to be squared)
- Number of each pattern instance in the game. (Ex: 3 blocks, 5 gliders and 6 LSS).

The main program should then create sub-processes and then pause the game according to the defined frequency to save a snapshot to disk. (a snapshot is a representation of the grid at that moment). Notice that the snapshot should be consistent, and the **barrier** concept should help you achieve that goal. At the end of the execution, you should save a final snapshot as well as display the information on screen.

Quality attributes

Your application must make use of:

(a) Multiple processes;

(b) Shared Memory;

(c) Memory Mapped Files;

(d) POSIX semaphores.

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

Notes

- 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 and **stop all processes** before exiting the Master process;
- Always release the OS resources that are not being used by your application;
- **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;
- 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 **November 11, 2012 (23h55)**