Operating Systems – 2023/2024

Informatics Engineering - ESTiG / IPB

Practical Work 2 (version 2023-12-18)

1 Goal

To parallelize a simulation of a 3-state version of Game of Life using the Process model, together with System V Shared Memory for data sharing and System V Semaphores for synchronization. The work is to be developed and executed in a Linux system.

2 Introduction

"The Game of Life [...] is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. The universe of the Game of Life is an infinite, 2D orthogonal grid of square cells, each of which is in one of two possible states: live or dead [...]. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent" [1]. A way to turn the game even more interesting is to consider a third state: "dying". So at each step in time of this new 3-state Game of Life the following transitions occur:

1. Any live cell goes to dying state in the next generation;
2. Any dying cell with two or three live or dying neighbours survives;
3. Any dead cell with three live or dying neighbours becomes a live cell;
4. All other dying cells die in the next generation. Similarly, all other dead cells stay dead.
5. In the initial state a cell is live or dead, not dying.

"The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed [...]. Each generation is a function of the preceding one. The rules are applied repeatedly to create further generations." [1]

3 Details

In this work you are required to implement the 3-state Game of Life in the C language. But, before that, start by understanding the base problem (read the Wikipedia page [1]). Important: notice that our set of rules was adapted to consider the extra state "dying".

Use as a starting point the supplied file `single-process-game.c`. This file already includes some .h header files (you may need more), defines characters to represent dead, dying or alive cells, defines constants to identify a specific or a random pattern, defines constants that are useful when using a random pattern, defines global variables for the number of lines and columns of the 2D grid, defines global pointers for the current generation and the next generation grids, and defines the maximum number (`NITERS`) of iterations for the game (in case the game does not first converge).

The same file also includes several empty functions to be filled (see comments) and one `clearScreen()` function already fully implemented. The later is to be used before showing in the screen the next generation of cells; in order to be able to use `clearScreen()` you must compile the program with `-lncurses`, for instance: `gcc single-process-game.c -o single-process-game.exe -lncurses`. Note: you may need to install first the ncurses library (`sudo apt-get install libncurses-dev`).

3.1 Single Process Version

Develop a sequential (single process) solution, where the main concern is to correctly implement the 3-state Game of Life algorithm, using the simple Oscilattor patterns [1]. Basically you must provide the missing code that implements the behavior described in the various comments scattered in the provided `single-process-game.c` file.

This first version should already be able to receive from the command line the kind of initial pattern to be used, with 0 to 4 meaning a certain Oscillator pattern, and 5 meaning a random pattern. Example of usage: `single-process-game.exe 0`.

Regarding the simple Oscilattor patterns: they may be easily fitted in 2D arrays (grids); but, in the C language, 2D arrays have a 1D arrangement in memory (one line after the other) and so you may simply use 1D arrays (char *) to store the grids, provided you do the correct math to access a cell at a specific line and column.

Besides supporting the simple Oscilattor patterns, your solution must also support an initial `RANDOM` pattern. In this case, the number of lines and columns are defined by the constants `NLINES_RANDOM` and `NCOLUMNS_RANDOM`, whose values (50 and 200) are appropriate to execute the game in a maximized terminal (but you may change them). Also, when randomly generating the initial pattern, you must ensure that the initial number of alive cells is (approximately) 10% of the total number of cells.

3.2 Multi Process Version

Once you have developed the previous Single Process Version, and that version is ensured to be working correctly, you should derive from it a new version where several processes divide among them the work performed in the `gridEvolve()` function. This new version will be placed in a file named `multi-process-game.c`.

Basically, you will now have a parent process and a number of `NWORKERS` children (`NWORKERS` is a constant whose value you may change in the program to define its degree of parallellism). It will be up to the parent process to show in the screen the initial grid, and each of the newly generated grids. It is up to the children to generate the next grid based on the previous one. So, you have to carefully synchronize the parent and the children: the parent must wait for the children to end producing a generation before it can show it in the screen; the children should only start producing the next generation after the parent has shown the previous one. This synchronization must be implemented with **System V Semaphores**.

Also, the work involved in generating the next grid must be divided among the children in the most possible balanced way. Basically, the grids must be split in `NWORKERS` slices (i.e., one slice per child process) of equal dimensions (in case the grids are not evenly divisible by the children, just choose a specific child to handle the remainder of the division). To share the grids (and other global data structures) among all the processes involved you must use **System V Shared Memory**.

Compare the results of the Multi Process Version with those produced by the Single Process Version. They should be the same for the same initial conditions (note that to compare the results of the `RANDOM` pattern you must ensure the same seed parameter is passed to the function that initializes the random number generator).

4 Groups and Deadlines

The work must be done in groups of 2 students (exceptions only allowed for working students or other authorized cases). Please try to keep the same group of Work 1, unless you made that work alone, in which case you should try to form a group.

The deadline for submission is **9am January 15th 2024**. Submissions should be done primarily through the Atividades / Assignments module of the Operating Systems area in the virtual.ipb.pt site, as a single ZIP file with all the deliverables. If (and only if) the site is unavailable, send the ZIP file by email to `arnaldo@ipb.pt` no later than 30 minutes past the deadline. The deliverables are the source code files `single-process-game.c` and `multi-process-game.c` (do not submit any executables, or your submission may be blocked in case you need to submit by email).

The defense (demonstration and oral evaluation) of the work will be carried out on a date to be announced in due time. The classifications of the elements of each group may be differentiated, based on individual performance during the defense.

5 Evaluation

The evaluation of this practical work will be based on the analysis of the code of the files `single-process-game.c` and `multi-process-game.c`, and on the live demonstration of their execution (no report is needed for this work, but you may supply explanatory notes if you want).

6 References

[1] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life 3