
# Operating Systems: Project #2

## Introduction

This assignment will focus in 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 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 concurrent processes
- Map a file on disk into a buffer in memory and use it to share data between concurrent processes
- Use the library `<semaphore.h>` to create and use semaphores to synchronize the access to shared resources from competing processes

## Support Material

- K. A. Robbins, S. Robbins, "Unix Systems Programming: Communication, Concurrency, and Threads", Prentice Hall:
  - *Capítulo 14 – "Critical Sections and Semaphores"*
  - *Capítulo 15 – "POSIX IPC" (secções 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
    - POSIX Semaphores: `<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:
  - Notes on Unix basic commands
  - Notes on reading and writing to files
  - Notes on `Makefile`

## Project Description

The goal of this project is to **build two versions of a concurrent program for performing the horizontal inversion of image files**. Both programs use a set of collaborative processes to concurrently transform an image file. In order to allow these processes to communicate among them, each version of the program will make use of a different technology. One version will employ **shared memory (SM)**, while the processes in the other version will communicate by means of **memory mapped files (MMF)**.

As in the previous assignment, we provide the sequential version of the desired application. To execute this program, you must provide two parameters: the filename of the source image and the filename for saving the inverted image. The program creates the inverted image and displays the time taken by the conversion. Additionally, it also displays header file information (i.e., image height and width) and information about the inversion process (for each line). Figure 1 illustrates the usage of this software.

```
nuno@athena:~/project2$ ./invert sketch.ppm out.ppm
Opening input file [sketch.ppm]
Opening output file [out.ppm]
Getting header
set type as P6
read next header value
read next header value
read next header value
Got file Header: P6 - 320 x 240 - 255
Saving header to output file
Starting work
Reading row... Got row 1 || Inverting row... Done || Saving row... Done
Reading row... Got row 2 || Inverting row... Done || Saving row... Done
...
Reading row... Got row 240 || Inverting row... Done || Saving row... Done
Cleaning up...
Closing file pointers.
Time ellapsed to complete task: 196345.000 microseconds
Time ellapsed to complete task: 196.362 miliseconds
Done!
```

Figure 1 – Sequential conversion tool usage.

To make things more exciting, we propose a challenge: you must optimize the two versions of the program, the one using MMF and the one using SM, and make them run faster than the sequential version in a multi-core processor. You must design your programs in a way that minimizes any shortcomings, or performance penalties, imposed by the two technologies.

Use your creativity and create the most efficient design you can. But, make sure that your plan includes: 1) Multiple concurrent processes; 2) Shared Memory/Memory Mapped Files; and 3) POSIX Semaphores.

To fulfil the objectives of this assignment you can be as creative as you want, given that the OS functionalities identified above are used in your solution. Nonetheless, to offer you a head start we describe a possible architecture for this software. You are free to decide if you want to use this design, or not.

In this solution, there is one Master process that creates several Worker processes that, in turn, cooperate to read and write the image data and produce an inverted copy of it. Each worker is responsible for processing one line of the image at a time. The number of lines in each image is available on the image header. This scenario is illustrated in Figure 2 and further discussed in the following paragraphs.

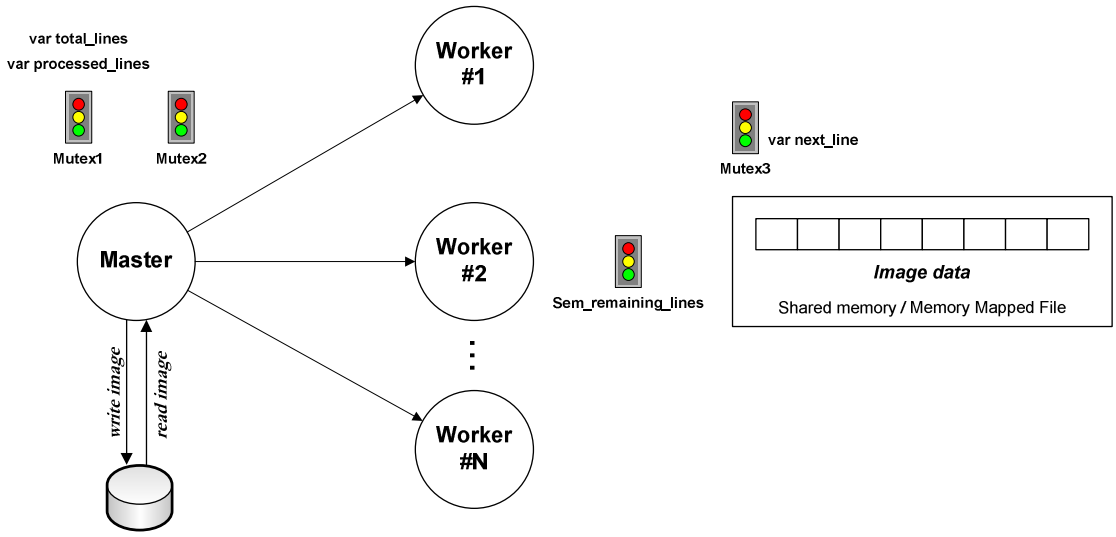


Figure 2 – The multi-process image converter tool.

The **Master process** receives the source and destination filenames from the command line. Next, it uses the received data to open the source image file and collect any relevant information about the image, such as the number of lines. Afterwards, the Master process creates the shared memory segment (or memory mapped file) that will hold/map the source image data.

The Master creates the Worker processes. The right number of workers to setup varies according to the file size and number of cores available in the computer. Later you can execute some tests to understand what should be the appropriate number of workers. The Master process takes note of the current time and sets the `Sem_remaining_lines` semaphore's initial value to match the number of lines in the image, which will be processed by the workers. Afterwards, the Master blocks on `Mutex2` and waits for the workers to complete the inversion of all lines in the image.

Each **Worker process** starts by waiting for the `Sem_remaining_lines` semaphore. When allowed to proceed (i.e., there are image lines to be processed), the Worker waits for `Mutex3` in order to be able to read and update the value of the variable `next_line`, which represents the next image line to be processed. After posting in `Mutex3`, the Worker inverts the line it selected (note: check the function to invert lines of images available in the code provided). When this task is completed, the Worker waits for `Mutex1` in order to increment the number of lines processed (variable `processed_lines`), compares the current number with the total number of lines to be processed (variable `total_lines`), and frees `Mutex1`. The goal of this verification is to understand if this particular worker is the last one to execute (i.e., the image has been completely inverted). If it is the last one, it must wake up the Master process (by posting on `Mutex2`). The Master displays the total elapsed time in these operations to the user and performs cleanup and shutdown operations.

Note that, on the Master process of the SM version, you have to explicitly write the inverted image to disk from the final data on the SM segment. The structure of the SM segment must be defined by you. On the other hand, the MMF version will first have to create a copy of the image (and rename it to the destiny filename) and then execute the inversion on that copy.

**Speedup:**

After coding and testing your applications, you should compare each concurrent version (SM and MMF) with the sequential version, in order to calculate the speed gain. Speedup is a metric defined by the following formula:

$$S_p = \frac{T1}{Tp}$$

Where $Sp$ is the speedup obtained when executing the program in a system with $p$ cores or CPUs, $T1$ is the execution time of the sequential version of the program, and $Tp$ is the execution time of the concurrent program in a machine with $p$ cores or CPUs.

You should also compare the times obtained with the SM version and the MMF version. For this comparison, simply replace $T1$ and $Tp$ with the times obtained with these two versions of the application.

**Important Notes:**

- Plagiarism or any or kind of fraud will not be tolerated. Attempts of fraud will result in a 0 and consequent reprobation 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!
- Use a `Makefile` for simplifying the compilation process.
- Always release the OS resources that are not being used by your application.
- 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 all elements of the group).
2. Any external libraries must be provided along with the source code file, with usage instructions. Please include a `Makefile` for simplifying the compilation process. Use a ZIP file to archive multiple files, if necessary.
3. The project should be submitted via http://moodle.dei.uc.pt.
4. The submission deadline is **November 15, 2010**.