# 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, mutexes 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, mutexes and condition variables to synchronize the access to shared resources from 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

    - Manual pages of `sem_post`, `sem_wait`, `sem_open`, `sem_init`.
    - Manual 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`

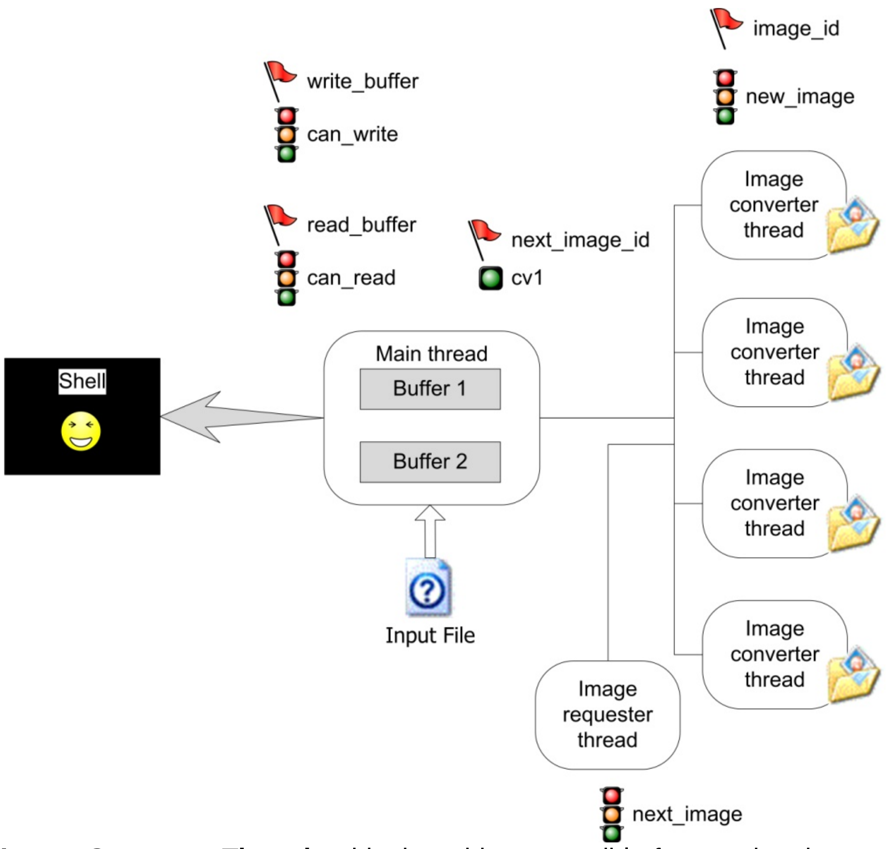
## Project Assignment

### Overview

In this project you have to develop a program to make “ascii movies” for the Linux shell. This means that you have to convert several images to ascii strings and show them on the shell, in a predefined order and rhythm, thus creating an animated movie. Each ascii picture represents a frame in the video. Your objective is to achieve the highest frame rate possible (fps – frames per second).

The program must be written as a multi-threaded process. As shown in the following figure, there three kinds of threads:

1. **Main Thread** – responsible for reading an input file in which the order for inserting images into the “movie” and the identification of the images to use is defined. This thread creates the Image Requester Thread and all the Image Converter Threads (the number of threads to create is undefined.) After these initial tasks, the Main Thread starts to execute in loop and, in each iteration, it empties to the screen the buffers of ascii strings it holds in a ordered fashion, one at a time and only when they hold fresh information (each buffer is emptied only once for each time it is written.) It empties the two buffers alternately. This thread must also calculate and show on the screen the number of times it can empty both buffers per second, this is the same as computing the Frame Rate in Framer Per Second (FPS).
2. **Image Requester Thread** – this thread is responsible for sending requests for each image, together with frame order information (in the movie), to the Image Converter Threads. It executes in a loop and creates a new request, per iteration, using the chosen image identifier. This identifier can be a filename, a path, or anything else that you would like to use.



3. **Image Converter Thread** – this thread is responsible for creating the ascii strings that represent each one of the images for the “ascii movie”. Each thread handles only one image at a time, they receive the image identification from the Image Requester Thread, transform the target image into a ascii string and write the string into one of the buffers used by the Main Thread. The buffer used must be empty and must not be in use in the

Main Thread. Also, the frame number on the image must coincide with the expected frame order on the Main Thread. This means that, if a Image Converter Thread tries to write a frame into the buffers which as a order number different from the one that is expected on the Main Thread, it must wait until it the numbers coincide and let another thread check for its opportunity.

IMPORTANT NOTE: No caching mechanisms of ascii strings are allowed on the program.

**Synchronization**

The synchronization rules on this program can be addressed by the usage of 4 semaphores, one condition variable, and an undefined number of mutexes.

- Semaphore **next_image** – initially this semaphore has the value 1. To emit a new image request the Image Requester Thread must be able to decrement the semaphore. The Image Converter Thread signals these semaphore after a successful reading of the requested image ID.
- Sempahore **new_image** – initially holds the value 0. The Image Requester Thread signals this semaphore in order to "wake up" one of the Image Converter Threads after updating the **image_id** variable which holds the identification and frame order of the next image to convert.
- Semaphore **can_write** – initially holds the value 2. The Image Converter Thread on a successful *wait()* on this semaphore checks which is the buffer to write into (**write_buffer**), writes the string into the buffer, updates the id of the buffer to write next and signals the **can_read** semaphore.
- Semphore **can_read** – initially holds the value 0. The Main Thread knows that it can empty one of the buffers (identified by **read_buffer**) when it can decrement this semaphore successfully. There must be a time out on this *wait()* action to allow the Main Thread to "wake up" any Image Converter Threads that might be "sleeping" on the CV1 condition variable while waiting for their turn to write into the buffers. After a successful reading the Main Thread updates the **read_buffer** variable.
- Condition Variable **CV1** – The Image Converter Threads use this variable to know if it is their turn to write into the buffers. Whenever a thread checks this variable, despite being authorized to write or not, it must assure that any thread that is waiting to check the variable value is "waken up" and allowed to do so.
- Mutexes – Several segments of code in this program are required to execute in mutual exclusion. Be sure to comply with such needs.

In general:

- You must use the `main()` function to launch the N+1 threads and wait for their completion. If a SIGINT signal arrives at the program it should be handled by the main thread in order to shutdown the program and its threads a clean way. Consider the following example on how to accomplish this and consult the *Linux* manual on the `sigwait()` function:

```
sigset_t set;
int sig;
sigemptyset(&set);
sigaddset(&set, SIGINT);
pthread_sigmask(SIG_BLOCK, &set, NULL);

while (1) {
        sigwait(&set, &sig);
        switch (sig) {
                case SIGINT:
                        /* handle interrupts */
                        break;
                default:
                        /* unexpected signal */
                        pthread_exit((void *)-1);
        }
}
```

**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 **December 6, 2009**.