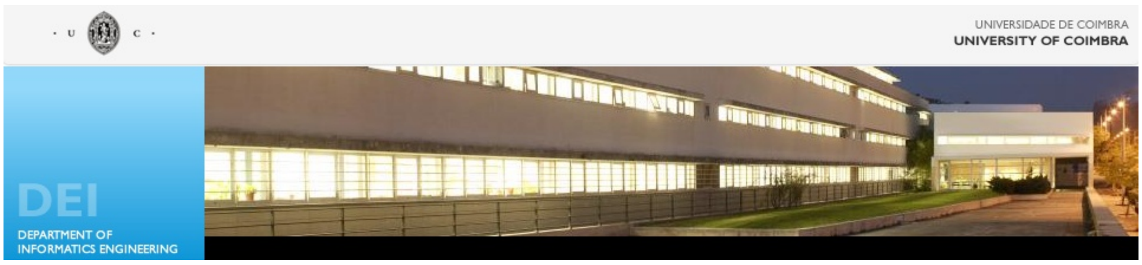


Operating Systems: Project #1

Introduction

This assignment will focus in the following tasks:

- creation and management of processes;
- inter-process communication using pipes;
- asynchronous signals;
- I/O multiplexing by using `select()`.

Processes

A process is an instance of a program that is executing. In this assignment the student will learn how to create processes and how to understand the UNIX process model.

Signals

A signal is a software notification of an event sent by the operating system to a process. A signal is generated when the event causing the signal occurs. Signals can be used by the operating system to report exceptions to running processes, such as the usage of an invalid memory reference or the occurrence of errors while accessing a file. Signals may also be used between processes to notify specific conditions. To guarantee the portability of an application, signals should be referred by their name. When you use the signal number it migh not be the same on different operating systems. Every signal has a custom default treatment preset by the operating system. As an example, the signal `SIGTERM` will force the termination of a process. By using a signal handler, a process can override the default action associated with a signal, which is normally the termination of the process.

Pipes

Pipes are the simplest mechanism offered by the operating system for inter-process communication. A pipe is a communication buffer between two processes: it has two descriptors, one for writing other for reading. Write and read operations are done in a FIFO order (first-in-first-out).

There are two kinds of pipes: unnamed pipes and named pipes (also known as FIFOs).

- Unnamed pipes only allow communication between hierarchically related processes (parent and child processes);
- Named pipes allow the communication between any process. A special file is created in the file-system through which processes can write or read data. Named pipes are persistent over time.

I/O Multiplexing

When a process needs to handle multiple I/O events it should make use of the select() routine. This system call allows the process to block reading in a set of file-descriptors: when one of them has something to read the process is unblocked and conducts the read operation.

Objectives

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

- Create and manage processes in Unix, using C.
- Send, handle, ignore and block signals.
- Allow the communication between processes using pipes and named-pipes.
- Use `select()` to implement I/O multiplexing.

Support Material

- K. A. Robbins, S. Robbins, "Unix Systems Programming: Communication, Concurrency, and Threads", Prentice Hall:
  - Chapter 2 - *Programs, Processes and Threads*
  - Chapter 3 - *Processes in UNIX*
  - Chapter 4 – *"The select Function"* (section 4.4)
  - Chapter 6 – *"Unix Special Files"*(section 6.1 to 6.4)
  - Chapter 8 – *"Signals"*

- "Programming in C and Unix", available at

  http://gd.tuwien.ac.at/languages/c/programming-dmarshall/:

  - *Process Control: <stdlib.h>,<unistd.h>*
  - *Interprocess Communication (IPC): Pipes*
  - *IPC:Interrupts and Signals:*<signal.h>

- Online Unix manual

  - Manual page of signal().
  - Manual page of pipe().
  - Manual page of select().

- Course materials:
  - Notes on Unix basic commands
  - Notes on reading and writing to files
  - Notes on Makefiles

Project Assignment

Overview

The objective of this assignment is to write a concurrent parser for analysing IP addresses in the log files of an Apache Web Server. This parsing routine should count the number of accesses to our web server from a well-defined range of IP addresses (e.g., addresses between `78.24.0.0` and `78.24.124.124`).

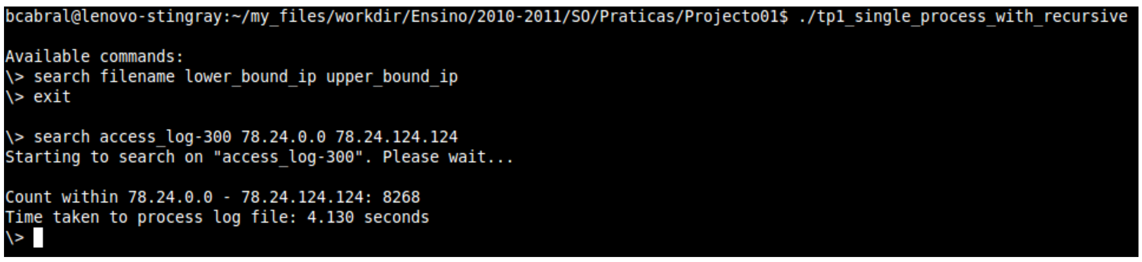
To make things more exciting, we propose a challenge: your program should be able to outperform the sequential version when executing in a computer with a multi-core processor.

Your application must make use of: (a) multiple processes; (b) signals to communicate events between processes; (c) pipes to communicate data between processes; (d) the `select()` routine to multiplex reads/writes in multiple pipes.

To make things simpler we provide the sequential version of the program along with this assignment. The program presents a shell environment to the user.

This shell is extremely simple and only accepts two kinds of commands: (1) `exit` – to abandon the program; (2) `search filename lower_bound_ip upper_bound_ip` – `search` is the command name, `filename` is the name of the log file to parse, `lower_bound_ip` is the inferior limit for the sequential list of IPs to search, and `upper_bound_ip` is the upper limit of the same list.

In the following Figure it is possible to observe the kind of output we get when executing this program. In the text we can see the number of accesses to our web server coming from the defined range of IPs and the time (in seconds) that took the search.



```
bcabral@lenovo-stingray:~/my_files/workdir/Ensino/2010-2011/SO/Praticas/Projecto01$ ./tp1_single_process_with_recursive

Available commands:
\> search filename lower_bound_ip upper_bound_ip
\> exit

\> search access_log-300 78.24.0.0 78.24.124.124
Starting to search on "access_log-300". Please wait...

Count within 78.24.0.0 - 78.24.124.124: 8268
Time taken to process log file: 4.130 seconds
\> 
```

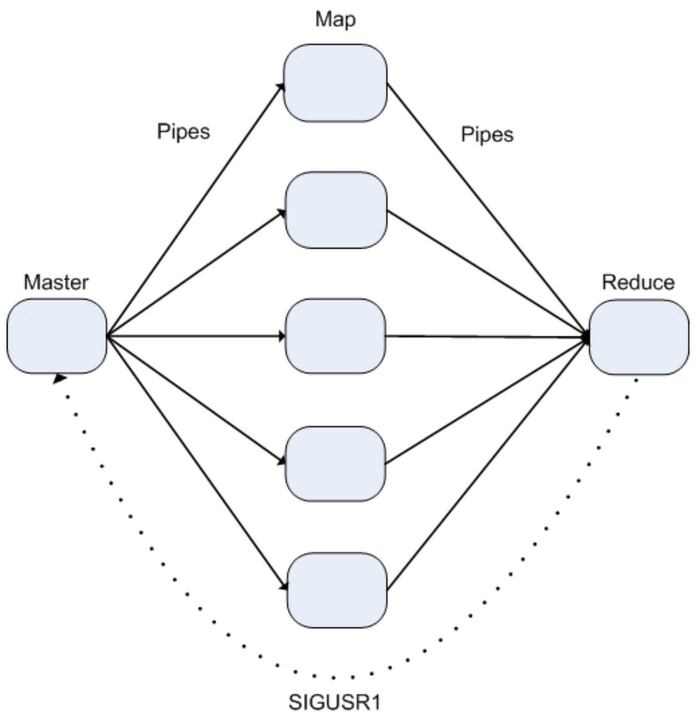
To fulfil the objectives of this assignment you can be as creative as you want, given that the OS functionalities identified above are used in your solution. Nonetheless, to offer you a head start we describe a possible architecture for this software. You are free to decide if you want to use this design,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

In order to distribute work by several concurrent processes, we based this solution in a Map-Reduce architecture. Thus, in the system there are three kinds of processes:

- The *Master* process – it is the shell environment. It reads the user input and creates the processes to execute the necessary tasks for parsing data and collecting results. After starting a new workload (time starts counting), the shell blocks until the final result is calculated. At this point, the process receives a signal (`SIGUSR1`) and stops the clock. Then it displays the total time on screen and is ready to accept more commands.
- The *Map* process – it parses the log file searching for a subset of the IP addresses in the list. After counting the existent matching occurrences, the Map process communicates its results to the Reduce process and pauses while waiting for more work.
- The *Reduce* process – It waits for results from the Map processes, sums up every parcel to obtain the grand total and presents it on screen. Finally, it sends a signal (`SIGUSR1`) to the Master in order to stop the clock.

Communication between the Master process and the Map processes is done through several pipes: one pipe per Map process. Between the Map processes and the Reduce process, there are also multiple pipes, one for each Map process. Thus, the Reduce process needs to use `select()` to receive data in all pipes at the same time. The Reduce process syncronizes with the Master process by using signals. The messages (data) passed between processes should contain all the necessary information to manage the execution (start/stop/pause) of any processes in the system.

The following Figure illustrates the architecture that we have just described.



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.

Important 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 or kind of fraud will not be tolerated**. Attempts of fraud will result in a 0 and consequent reprobation in the OS course.

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. 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 **October 18, 2010**.