Operating Systems Project

Pablo Rodriguez - 2014084649

Diego Solís Jimenez - 2014027244

Geovanny Espinoza Quiros - 201404508

Introduction

How do threads work?

Threads are instances of the same process, this can be considered as task. In a multithreading compatible Operating System there is a scheduler that is in charge of deciding the order and time of execution of each thread. **[2]** Threads, differ from processes in the sense that threads from the same process share: **[2]**

- Text segment (instructions)
- Data segment (static and global data)
- BSS segment (uninitialized data)
- Open file descriptors
- Signals
- Current working directory
- User and group IDs
  Yet, the system identifies each of the threads using:**[2]**
- Thread ID
- Saved registers, stack pointer, instruction pointer
- Stack (local variables, temporary variables, return addresses)
- Signal mask
- Priority (scheduling information)
  There are several scheduling algorithms that intent to give each thread a fair share of processor time interleaving each one. **[2]**

How are threads implemented

Threads are implementing by creating a thread and assigning it a task to complete. This way it will be scheduled rather than be executed in a serial way.
It’s important to mention that in Linux, threads are not something special, they are treated as any other process, meaning that there is not a concept of threads.
In Linux threads are implemented by “forking” process which creates a child process from a parent one and this is scheduled by the OS as any other process. Thi is achieved by using the system call: fork(), which does the fork and returns the process id for that “thread”. **[3] [4]**

Overview of scheduling Mechanisms to be implemented

FiFO (First In, First Out): This scheduling algorith sends the processes to execution in the order they arrive and gives them the CPU until they are finished executing. This of course brings several problems, because shorter tasks will be held back by longer tasks that came before, also, it's imposible to implement a real time scheduling system by using FIFO since any real time process that comes during the execution of another will have to wait preventing it from executing as it comes (real time) this as an example of the implications it brings. **[1][5]**

Selfish Round Robin: This is a mix between FIFO and common Round Robin. This implements two queues for processes that arrive, adn two constants of priority increase, one for each queues. For the ready queue there is "a", and for the accepted queue there is "b", with every tick the priority of any process in each queue will be increased by it's respective value. **[6]**

Ready Queue: Any new comer process will be asigned to this queue with a priority of 0. Then it's priority will be increased by "a" with each "tick". When it's priority becomes the same or superior to the processes in the Accepted Queue the process will be sent to the end of said Queue. **[7]**

Accepted Queue: The first frocess to come will be asigned instantly to this queue, then it's priority will start to be increaded with every "tick" by b. The

processes in this queue are executed using common Round Robin algoritm. **[7]**

Lottery Scheduler: In Lottery Scheduler "tickets" represent the "priority" of a process. When processes enter the scheduling they are given a certain number of "tickets" according to their priority then a "lucky throw" happens and chooses a ticket to give the processor to, the process that has this ticket is given the CPU to execute during certain amount of time. It's important to mention that the tickets are given from a total number (commonly portrait a consecutive numbers from 1 to the total amount of tickets). **[8]**

This method provides three mechanisms to improve it's functionalities:
Ticket Currency*: The user can give his tasks a certain amount of tickets locally to set the priority of them in relation with one another, then those get converted into global currency and scheduled with the rest of the processes.*
Ticket Transfer: A task can transfer its tickets to another task to maximize its performance.
* **Ticket Inflation**: With this mechanism a process can increase its number of tickets. This ins only used in an eviroment where processes trust one another. **[8]**

Real Time Scheduler: In real time schedulers there are deadlines for each process, this means that it has to be finished before this deadline arrives. This is done by defining priorities acording to the proximity of a process's deadline. There are two kind of deadlines in this scheduler: **[9]**

- **Soft**: The deadline is not so important, allowing for certain amount of "overdue" time for the execution of the task.**[9]**
- **Hard**: The deadline must be respected and the task must be completed accordingly (before it arrives).**[9]**

This scheduler also defines processes in two categories:**[9]**

- **Aperiodic**: Processes that may happen at any given time or may not.
- **Periodic**: Processes that happen on a regular basis.

Both categories have in common that they must be finised in a certain amount of time, but in the case of the periodic kind they must be finised before the next periodic process happens.**[9]**

About Docker containers

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. **[6]**

About Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

About POSIX pthreads library

Project brief specification (what does it do and how does it)

The project consists of several web-servers developed in C language that must be executed in docker containers in an Azure virtual machine running CentOS as daemons, which must implement different scheduling algorithms. This is archived by creating our own thread library "mypThreads"(inspired from pThreads) which provides the methods for thread managing, meaning this also provides the characteristics to be used by the scheduler to archive the desired result.
The system must tolerate creating threads for each scheduler, this by overriding methos and allowing for different parameters to be set in the functions.

The server versions must be:
Sequential Version*: This web server takes care of serving one request at a time, following a FIFO policy.*
Forked version: Every time that the web server gets a request , a heavyweight process is going to be created using fork, this process is going to serve only this request (when it's done with the request, the process has

to die, ensure the process descriptor it's properly destroyed and the forked processes are not left as zombies). The main process will be waiting for the next request to be served.
* **Threaded version**: This one is similar to the one mentioned in the last point but instead of creating heavyweight processes, you will create a thread (light-weight process) to take care of each request. The thread will be terminated when the request is completed.

- **Pre-fork version**: The web server is going to receive an argument in the corresponding configuration file that is going to indicate how many processes, which will be called workers, should be created in advanced using fork in order to take care of all the different requests. The main process is going to take care of listening the socket and assign the work to any available worker. Each worker will be managing the request that was assigned to it and then is going to be blocked while waiting for any upcoming requests. It is also required to create an elegant mechanism (some internal command that will be passed to all the workers) to deactivate all the web server (ensure no zombies are left behind) which executing the stop of the daemon.

- **Pre-threaded version**: The web server is going to receive an argument in the corresponding configuration file that is going to indicate how many worker processes should be created in advanced using mypthreads in order to take care of the requests. The main process is going to be listening over the socket and assigning the requests to the different workers that are available. As soon as each worker finishes its request it will be blocked while waiting for any upcoming requests. It is also required to create an elegant mechanism (some internal command that will be passed to all the workers) to deactivate all the web server (ensure no zombies are left behind).

Development Enviroment

We are developing all the project using the languaje C. For the web server we are using the http protocol and sockets. For the case of the services we chose systemd. Azure is used as a computing platform.

Azure Setup

Create a Ubuntu Server 18.04 Virtual Machine. Then clone the project into the VM:

```
git clone https://github.com/paroque28/simple-webserver.git
cd simple-webserver
cd Program
```

Install docker-CE and docker-compose

```
bash -x ./install_docker
```

To start all the webservers run

```
bash -x start_webserver.sh
```

To stop all the webservers run

```
bash -x stop_webserver.sh
```

To create a test file of 500Mb:

```
 dd if=/dev/zero of=test.iso count=500 bs=1048576
```

Program Design

BenchMarks

The code that is attached in the source package include the first release of the benchmark, this program already estimate the total execution time and it can run the amount of threads and cycles that the user insert in the command line, another functionality is that the program can show the response time. **[10]**
A problem that had to be solved was to be careful with the thread ID

because since the thread is created must retain the same id for the ncycles and depending on how the thread is created, this id change at the moment that the function ncycles is called. Each thread has a struct that store some important feasures, for example, the domain and path that need to be consulted on the server, this data is really helpful to analyze the behavior of the threads for when they overlap and in this way we have knowledge at any time about their status.
For this version, the results are displayed in the terminal, but they are not yet in CSV format.

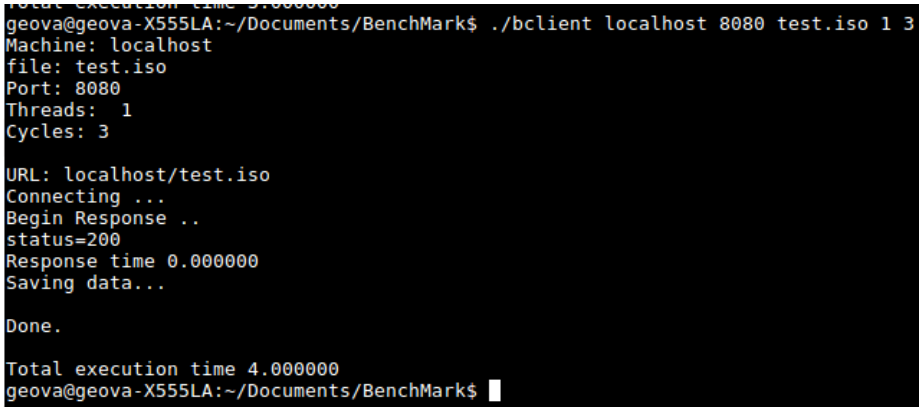

Fig 1. BenchMark Example 1

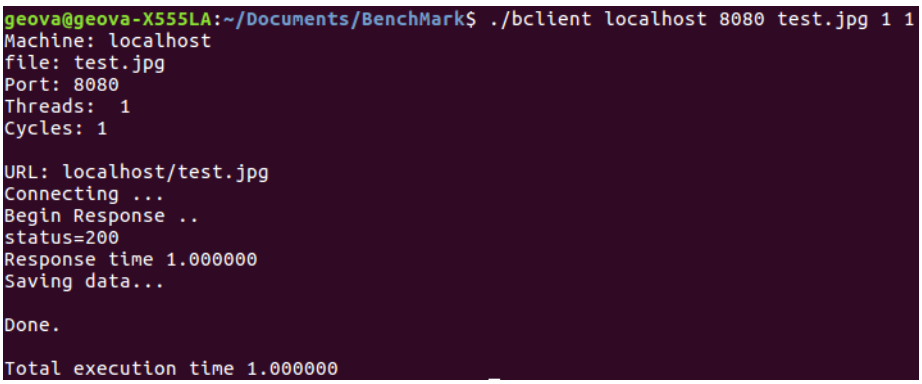

Fig 2. BenchMark Example 2

Machine	local host
file	test.jpg
Port	8080
Threads	4
Cycles	4
URL	localhost/test.jpg
Response time T1-C1	0.000182
Bytes received T1:	6226
Response time T1-C2	0.000154
Response time T1-C3	0.000157
Response time T1-C4	0.000132
Response time T2-C1	0.000187
Bytes received T2:	6226
Response time T2-C2	0.000084
Response time T2-C3	0.000086
Response time T2-C4	0.000067
Response time T3-C1	0.00008
Bytes received T3:	6226
Response time T3-C2	0.000101
Response time T3-C3	0.000083
Response time T3-C4	0.000086
Response time T4-C1	0.000078
Bytes received T4:	6226
Response time T4-C2	0.000063
Response time T4-C3	0.000089
Response time T4-C4	0.000079
Total execution time	0.007535

Fig 3. BenchMark Example of final release

Instructions on how to use the program

To run the diferent servers execute:

```
start_webserver.sh
```

To stop the servers execute:

```
stop_webserver.sh
```

Students Activity Logs

TimeTable for Pablo Rodriguez

Activity	Date	Hours
Create Docker Container	05-03-19	2
Create Docker Compose Hierarchy	19-03-19	2
Create Fork Version of Webserver	23-03-19	3
Fix Docker Compose Error Attach	27-03-19	2
Document progress	28-03-19	3
Research UserSpace Threads	30-03-19	2
Research Use of Context get and set	31-03-19	2
Research Use of timers get and set	1-03-19	2
Code Pthreads library + simple scheduler	2-03-19	4
Total hours:	--------	22

TimeTable for Geovanny Espinoza Quiros

Activity	Date	Hours
Review and Analyze the specification document	22-3-19	2
Benchmark and CSV research	23-3-19	1.5
Create and exampSle of benchmarks: Sequential-Forked-Threaded	24-3-19	2
Create more examples of benchmarks: Pre-fork and Pre-threaded	25-3-19	1
First phase document elaboration	26-3-19	1.5
Build the first benchmark code release	26-3-19 / 27-3-19	8

Activity	Date	Hours
Build the second and final benchmark code release	29-3-19 / 2-4-19	8
Research how to build threads without POSIX pthread library	2-4-19 / 3-4-19	5
Understand how a found library works to implement one	4-4-19 / 5-4-19	2
Second phase document elaboration	5-4-19	1
Total hours:	--------	32

TimeTable for Diego Solís Jiménez

Activity	Date	Hours
Review and Analise the specification document	22-3-19	2
Leftover bugs from ShortAssigment1 fix	23-3-19	4
Leftover bugs from ShortAssigment1 fix	26-3-19	4
First phase document elaboration	26-3-19	2
Second phase document elaboration	04-4-19	3
my_pthread library and threaded webserver testing	05-3-19	3
Total hours:	--------	18

Project Current Status

- [x] FIFO version of webserver
- [x] FORK version of webserver
- [x] Threaded version of webserver
- [ ] Pre-Thread version of webserver
- [ ] Pre-Fork version of webserver
- [x] My-Pthread Create Threads
- [ ] My-Pthread Mutexes

- [x] Docker containers of webservers
- [x] Benchmark tool

Conclusions

- The implementation of scheduling is a delicate matter.
- Designing a thread library requires an extensive understanding of how the OS works.
- Multithreading causes a significative performance improvement compared.

Recomendations and Suggestions

- Use Docker built in tool `docker-compose` for a simpler deploy of containers.

References

[1] First Come First Serve. (2019). Retrieved from http://web.cse.ohio-state.edu/~agrawal.28/660/Slides/jan18.pdf

[2] Krzyzanowski, P. (2014). Threads. Retrieved from https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html

[3] Implementing threads :: Operating systems 2018. (2018). Retrieved from http://www.it.uu.se/education/course/homepage/os/vt18/module-4/implementing-threads/

[4] Love, R. (2019). The Linux Implementation of Threads | Linux Kernel Process Management | InformIT. Retrieved from http://www.informit.com/articles/article.aspx?p=370047&seqNum=3

[5] First Come First Serve(FCFS) Scheduling Algorithm | Studytonight. (2019). Retrieved from https://www.studytonight.com/operating-system/first-come-first-serve

[6] What is a container (2019) Docker. Retrieved from https://www.docker.com/resources/what-container

[7] Casha, O. (2019). Selfish Round Robin. Retrieved from https://www.um.edu.mt/__data/assets/pdf_file/0010/110008/Multiprocessing.pdf

[8] Dusseau, A., & Arpaci, D. (2019). Scheduling: Proportional Share. Retrieved from http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf

[9] Audsley, N., & Burns, A. (2019). REAL-TIME SYSTEM SCHEDULING. Retrieved from http://beru.univ-brest.fr/~singhoff/cheddar/publications/audsley95.pdf

[10] Milevyo, (2015). Http protocol in C. Retrieved from https://stackoverflow.com/questions/33960385/how-to-download-a-file-from-http-using-c