

Operating Systems Project

Pablo Rodriguez - 2014084649

Diego Solís Jimenez - 2014027244

Geovanny Espinoza Quiros - 2014046508

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 implemented 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. This 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 algorithm 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 impossible 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, and 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 assigned 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 process to come will be assigned instantly to this queue, then it's priority will start to be increased with every "tick" by b. The processes in this queue are executed using common Round Robin algorithm. [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 is only used in an environment 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 according to the proximity of a process' 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 finished in a certain amount of time, but in the case of the periodic kind they must be finished 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

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. [11]

Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process, because they share it. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which execute another process while waiting for I/O to respond. All threads within a process share the same address space. A thread is spawned by defining a function and its arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster. [11]

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 achieved 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 achieve the desired result.

The system must tolerate creating threads for each scheduler, this by overriding methods 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. This is the less efficient version because does not take advantage of all the software and hardware resources that the machine has, but also has some advantages, for example, the server does not have to take care about memory protection because just one process will be running at time.
- **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. The main advantage of this version is the ability to continue with other executions when a failure happen, in that case it is enough to re-execute the query and the server will allocate a new process that will response it.

- **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. For this mechanism we use the Signals SIGSTOP and SIGCONT to deactivate and activate the workers respectively, but this solution required to share memory between the process and it was not a trivial knowledge so we chose an alternative solution that also fulfill with the requirements, the workers wait until his father give them a job.
- **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 threads 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. This translates to the need of creating a way of managing the threads, putting them to wait (using some sort of mechanism) and assigning them work (the arguments).

Development Enviroment

We are developing all the project using the language 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 webserver run

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

To stop all the webserver run

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

To create a test file of 500Mb:

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

To create the WebServer run the following command:
make TYPE=THREADED THREAD_LIB=MY_PTHREAD SCHEDULER=RR
where SCHEDULER can be: RR SELFISH_RR LOTTERY RT_EDF
and TYPE can be FIFO, FORK, PREFORK, THREADED, PRETHREADED

To Run all the benchmarks run the script called "runAllBenchmarks"
Be sure the file has the permission to run, if not then execute the following command:
sudo chmod +x runAllBenchmarks

Program Design

BenchMarks

The code that is attached in the source package include the final release of the benchmark. A problem that had to be solved was to be careful with the thread ID because since the thread ID was not unique for this version, the results are displayed in CSV format.

<figure>

 <figcaption>Fig 1. WebServers port distribution</figcaption>
</figure>

<figure>

 <figcaption>Fig 2. Benchmark result example.</figcaption>
</figure>

Instructions on how to use the program

To run the different servers execute:
``bash
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	1-04-19	2
Code Pthreads library + simple scheduler	2-04-19	4
Fix current basic pthread implementation	15-04-19	4
Factorize Round Robin Implementation	15-04-19	3
Meeting	15-04-19	1
Selfish Round Robin Algorithm	16-04-19	5
Lottery Scheduler	16-04-19	4
Real time EDF algorithm + add timing	17-04-19	4
Meeting	17-04-19	1
Add Mutex init, lock and unlock	18-04-19	5
Help Geovanny with prefork	19-04-19	3
Meeting	19-04-19	1
Use mypthread with webserver	21-04-19	3
Update docker and makefiles	22-04-19	3
Add mutex trylock, pthread exit and detach	22-04-19	3
Total hours:	-----	63

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 examples 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
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
Correct the benchmark feedback given in the 2 pthase review	15-4-19 / 16-4-19	4
Virtual meeting	15-4-19	1
research about shared memory between process and handle signals	16-4-19 / 18-4-19	6
Virtual meeting	17-4-19	1
Creation of prefork mechanism	16-4-19 / 18-4-19	4
Last virtual meeting	19-4-19	1
Final phase document elaboration	22-4-19	2
Total hours:	-----	50

TimeTable for Diego Solís Jiménez

Activity	Date	Hours
Review and Analyse the specification document	22-3-19	2
Leftover bugs from ShortAssignment1 fix	23-3-19	4
Leftover bugs from ShortAssignment1 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
Thread implementation research	12-4-19	3
Pre-threaded server basic design	13-4-19	2
Virtual meeting	15-4-19	1
Pre-threaded server first version	15-4-19	8
Pre-threaded server error correction	15-4-19	4
Virtual meeting	17-4-19	1
Pre-threaded server error correction (again)	15-4-19	3
Virtual meeting	19-4-19	1
Pre-threaded server waiting mechanism re-implementation	19-4-19	3
Final Phase Documentation work	22-4-19	4
Total hours:	-----	46

Project Current Status

- ☒ FIFO version of webserver
- ☒ FORK version of webserver
- ☒ Threaded version of webserver
- ☒ Pre-Thread version of webserver
- ☒ Pre-Fork version of webserver
- ☒ My-Pthread Create Threads
- ☒ My-Pthread Mutexes
- ☒ Docker containers of webserver

- ☒ Benchmark tool
- ☒ Tests and Report Generation
- ☒ Automation of Tests
- ☒ Automation of WS generation and Containers setup
- ☐ Webserver using mypthread library

Detailed MyPthread Current Status

- ☒ mypthread_create
- ☒ mypthread_exit
- ☒ mypthread_yield
- ☒ mypthread_join
- ☒ mypthread_detach
- ☒ mymutex_init
- ☒ mymutex_destroy
- ☒ mymutex_lock
- ☒ mymutex_trylock
- ☒ mymutex_unlock
- ☒ mypthread_setsched

Conclusions

- A user thread library can be implemented using user context switch provided by the glibc library
- Designing a user thread library requires the use of a scheduler in order to select the right next thread.
- User threads are less reliable as they can be interrupted by external functions like TCP.
- Setting states to manage threads makes the scheduling much easier as it keeps track of possible scenarios.
- Threads are meant to run, so making them wait has a certain impact on CPU use.
- Threads can be made to wait for a change in a flag (heavy on CPU) or can be made to wait for a signal (much lesser CPU weight).
- The use of shared memory must be done with care.
- The implementation of multiple threads results in the need for a management mechanism.
- Using signals to stop threads prevents the heavy consumption of CPU done by busy-waiting.

- Schedulers are important for a correct and optimum use of resources.
- A bad implementation of a Scheduler can lead to bad performance and the over-delay of certain processes and threads.
- System-time is mandatory, so it must be decreased to the minimum.
- A benchmark consists of tests, a tool to run the tests, and reports of the results.
- POSIX's thread library provides a full set of tools to implement, control and manage threads in C language.
- Threads allow for a better execution time than forking processes.

Recomendations and Suggestions

- Use Docker built in tool `docker-compose` for a simpler deploy of containers.
- Use a single script to run all of the others (automation), for a smoother use.
- Use csv to show the results coming from thread is useful to get ordered results.
- Use curl for a quicker/simple way of making requests to a web server.
- Always protect memory when there can be two simultaneous accesses to this resource.
- Be carefull about zombies process in order to avoid memory waste.
- Write exceptions for any openfile instruction and for any memory access query to avoid the program can freeze the computer.

References

[1] First Come First Serve. (2019). Retrieved from <http://web.cse.ohio-state.edu/~agrawal.28/660/Slides/jan18.pdf> (<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> (<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/> (<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> (<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>
(<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>
(<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
(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>
(<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> (<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>
(<https://stackoverflow.com/questions/33960385/how-to-download-a-file-from-http-using-c>)
- [11]** García, F. (2000). POSIX Thread Libraries | Linux Journal. [online] Linuxjournal.com
(<http://Linuxjournal.com>). Available at: <https://www.linuxjournal.com/article/3184>
(<https://www.linuxjournal.com/article/3184>) [Accessed 23 Apr. 2019].