École Internationale des Sciences du Traitement de l'Information (EISTI)

GSI Project – Final report

Authors : Titouan Bion, Axel Martin et Thomas Viau
Second year engineering students at the EISTI (Pau).

Addressed to M. Juan Angel Lorenzo del Castillo

May 2, 2016

Contents

1 Introduction

As part of our Parallel Programming and Network courses, we were asked to develop a job scheduler. This document is the final report of this project.
A scheduler is used to assign a job — processes, threads, etc. — to machine hardware resources such as processors. Every Operating System works with its own scheduler in order to give the user the impression of fluidity even if tons of process are working *"at the same time"*[1].

First, we will offer you a fast overview of the scheduler and job parameters to introduce you to our scheduling strategy. Then we will elaborate this scheduling strategy. After that we will develop the implemented features and state problems we encountered. Finally we will explain how to use our program.

2 Job and Scheduler Queue parameters

2.1 Job parameters

A job will eventually have different **compulsory** properties provided by the user:

- Burst time: an unsigned integer representing the time in milliseconds the job will use for the CPU to compute the job
- Priority: an integer representing the user fixed priority, lowest priority to the lowest integer
- CPU load : an unsigned integer representing percentage of used CPU by the program (from 1 to 100)
- Command line : the program command line along with as much parameters as needed

2.2 Scheduler parameters

Despite the fact that our program could work perfectly with default values and use the best for the system, the Job Queue itself can have optional properties. These properties will be provided by the user:

- `Timeout`: the time after which the application will kill the job, if set to 0 there would be no timeout and no job would be killed
- `Number of cores`: the number of processors the scheduler can send jobs to, by default it will be set to 1.
- `Time Slice`: an unsigned integer representing the time in milliseconds between each priority check [1].

[1] See our Scheduling Strategy Time Slice section for more information: 3.1.2

3 Our Scheduling strategy

3.1 Scheduling method description

There are many methods and as many algorithms to achieve scheduling. We chose the method that prioritize the shortest burst time programs to make the Operating System more dynamic and provide the best user experience. This method is used by the majority of Operating Systems, as Windows and UNIX systems (Mac, Linux distributions).

3.1.1 Overall description

This method named **Fixed-priority pre-emptive scheduling** will always execute the job with the highest priority to ensure that the user can have what he wants first. A job not being prioritized by the user will have a *default priority*. In the case where there are no prioritized jobs and several default priority jobs, the first job that will be executed is the one with the shortest *burst time* — the time the job will use for the CPU to compute the job — to ensure the velocity of the method. We will also implement the concept of *aging* priority too. A process will not wait forever in an infinite loop. Indeed, the time elapsed since the job's declaration will raise its priority. For instance, a job with default priority, a burst time of 600ms which was declared for 10s, will be run **before** a job with default priority, a burst time of 550ms which was declared for 4s.

3.1.2 Time slice

The *time slice* concept is used in this method too. It means that a job will not hog the processor for a time longer than a time slice. A job can be suspended if a job with higher priority comes in and will be resumed after the termination of the latter. Actually, this *priority race* is checked each time slice.

3.2 Example of scheduling with our strategy

The following chart (Figure 1) shows how processes are assigned to cores of a processor. Here we have a duo-core processor with four processes:

- A → Burst time: 300ms ; declared for 3s ; priority: high
- B → Burst time: 100ms ; declared for 2s ; priority: high
- C → Burst time: 200ms ; declared for 2s ; priority: normal
- D → Burst time: 100ms ; declared for 2s ; priority: normal

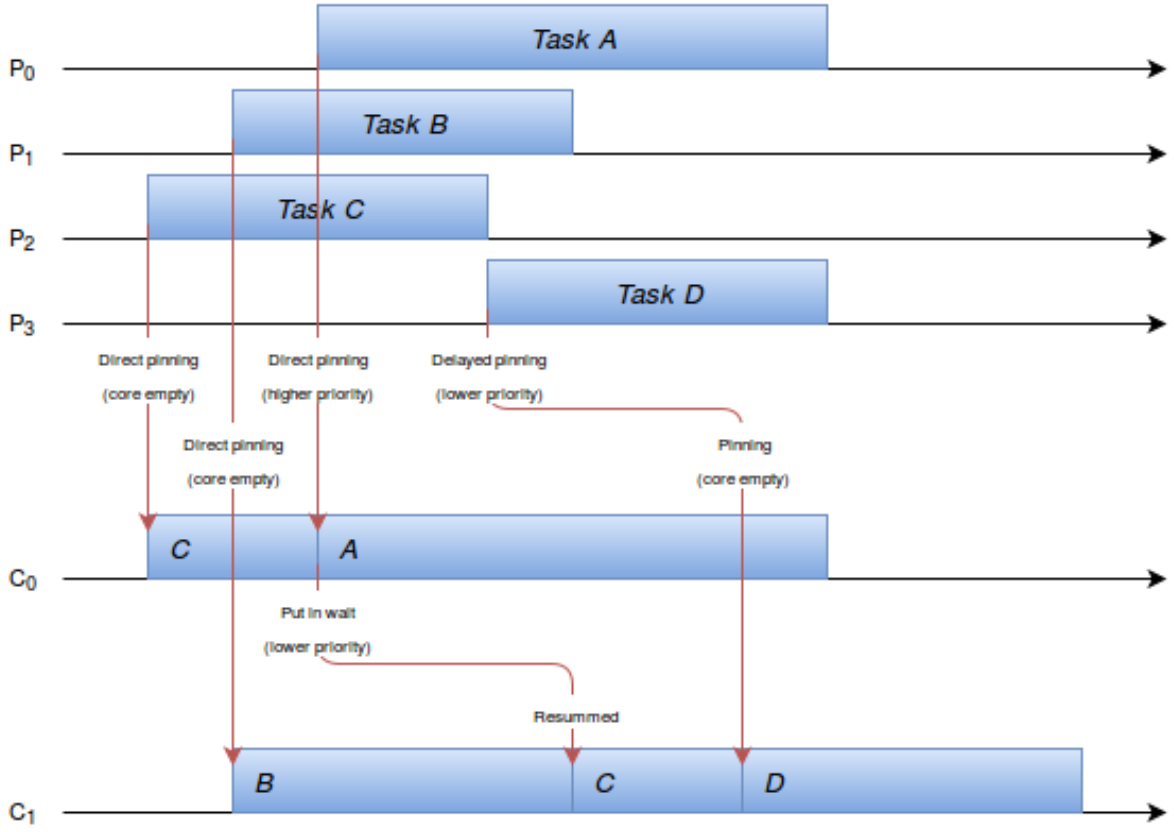

Figure 1: Example of scheduling following our strategy.

4 Implemented features

4.1 Sequential scheduler

We implemented a sequential scheduler that uses the scheduling method explain in 3. He owns a controller in charge of starting or stopping job in its Job Queue (see 4.2). At each Time Slice, the scheduler will command its controller to reorder its Job Queue according to our scheduling method.

File input format A file containing jobs can be specified to the scheduler, its controller will use it to fill its Job Queue. This file should respect the following format :

```
burst_time user_priority cpu_load command_line
```

with these types :

```
(unsigned) (int)          (int)     (std::string)
```

Here is an example of file :

```
120 5 10 sleep 3
120 5 100 program_toto -l -s 12
220 5 40 /bin/ls -la
120 5 20 sleep 12
```

4.2 Controller

4.2.1 Job queue management

The Job queue is reorder at each scheduler time slice. Its reordering follows our scheduling method keeping in mind that the priority is more important than the core management.

For example, we have a job with a priority of 10 and uses 75% of the CPU. We have also two other jobs with 50% CPU load and a priority of 9. In this case, the core won't be optimized and the first job will always be the most important to run.

4.2.2 Start/stop job

Starting a job To start a job, we fork our program. In the child, we execute the job command line and when the job is done we exit the child. In the father, we record the child pid to keep the possibility to stop it, resume it, etc.

Stopping a job To stop a job, we send a SIGSTOP signal to the child pid.

4.2.3 Core management

We chose to pin our processes until the core is full then skip to the next core. We could have chosen a method that equally distribute the charge between the cores. We will justify our choice with the following example.

Let's take this case, we have 4 processes which each take 25% of the CPU and we need to pin a 100% CPU process. Let's say we have 4 cores. We will now compare the two methods:

Equally distributed method inital state:

- CPU 0 : 25%
- CPU 1 : 25%
- CPU 2 : 25%
- CPU 3 : 25%

We can't pin the 100% process ! We need to wait until one job has finished with this method.

Our chosen method initial state:

- CPU 0 : 100% (25% * 4)
- CPU 1 : 0%
- CPU 2 : 0%
- CPU 3 : 0%

We can now pin our process to the second CPU and arrive in the following state:

- CPU 0 : 100% (25% * 4)
- CPU 1 : 100%
- CPU 2 : 0%
- CPU 3 : 0%

4.3 Dynamic feed of the Job Queue using `schedclt`

A tool `schedclt` — for scheduler client — has been developed in order to interact with the scheduler. This tool is launched without parameters and shows a prompt waiting for a Job string. After the user inputs its Job string, it will send it through a socket to the scheduler server. The latter will then announced to its Controller that it needs to inject the job and reload its job queue. You can see an illustrated version of this routine on Figure 2.

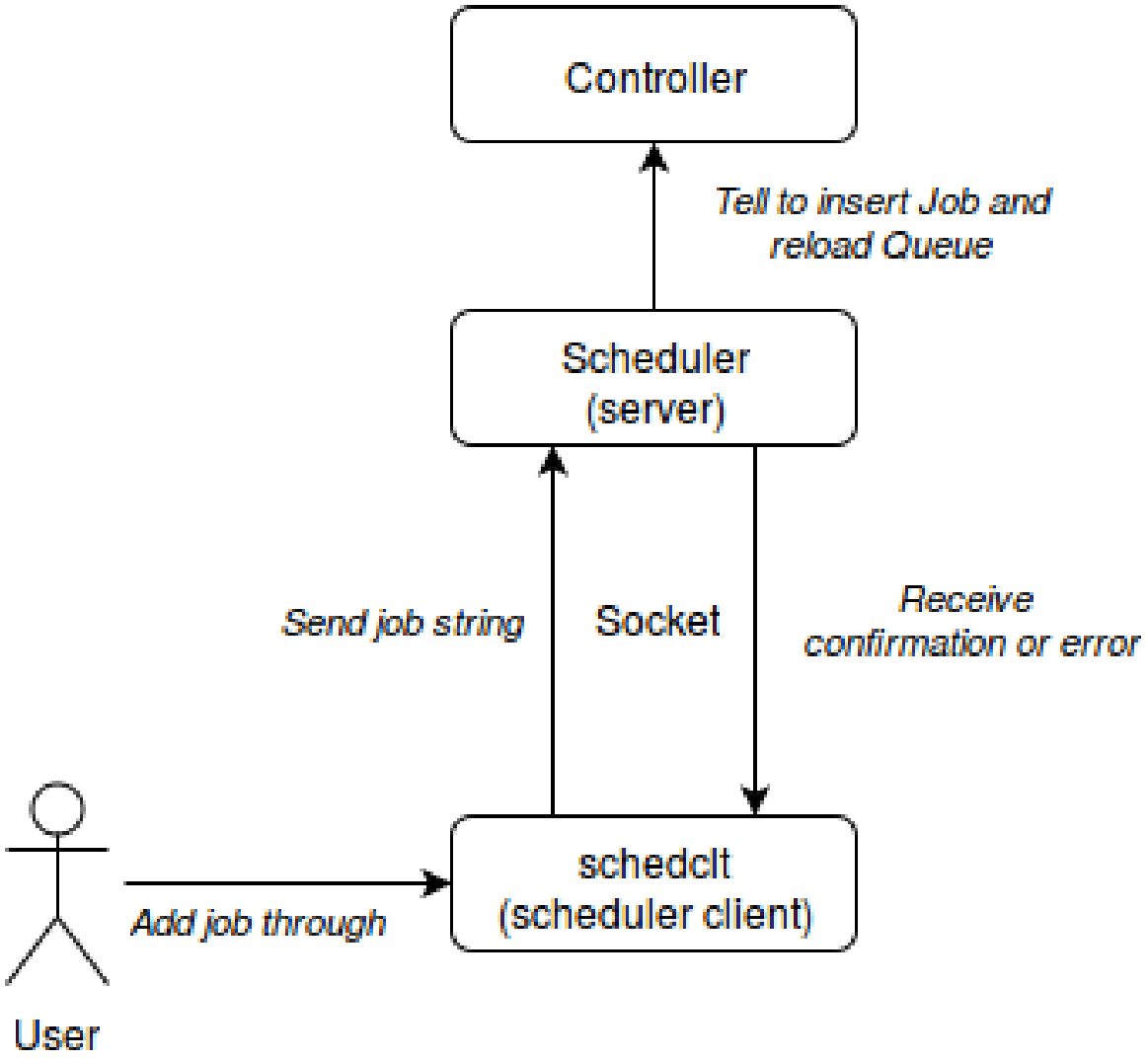

Figure 2: Scheduler client workflow chart.

5 Faced difficulties

5.1 OpenMP and MPI optimizations

We didn't have enough time to implement a parallel and distributed version of our scheduler. Reinforced by the lateness of course, especially on MPI, we didn't have enough time to think our architecture to implement a correct distributed version.

5.2 Language difficulties

As we were still having C++ courses during the development of this project, we made a large amount of bad architectural choices. We often wasted our time doing huge compulsory refactorings of our application.

6 How to use our program ?

6.1 Programs description

Our scheduler program is both responsible of scheduling and listen to a port as a server. This server can receive jobs from clients in order to feed the job queue as explain in 4.3.

Our scheduler program has the following parameters :

```
$ ./scheduler -h

Generic options:
  -h [ --help ]                    produce help message

File options:
  -i [ --input-file ] arg          input filepath containing jobs

Queue options:
  -s [ --time-slice ] arg (=100)   time elapsed between too priority check in
                                   milliseconds
  -c [ --core-number ] arg (=1)    number of cores used by the scheduler
  -t [ --timeout ] arg (=0)        maximum execution time in milliseconds of a
                                   job, if set to 0, no timeout will be
                                   used
```

The specified file should be formatted as it is explain in 4.1.

6.2 Sending Jobs with `schedclt`

To send jobs to our scheduler, we need to launch the scheduler program :

```
$ ./scheduler -s 2000 -c 4 -i ../dataset/jobs
```

In this example, we launch our scheduler with a 2000ms TimeSlice, 4 cores and Job Queue initialized with jobs contained in `../dataset/jobs` file.

Then, in a different terminal, we need to launch a client :

```
$ ./client
Enter your job string (--help for format & example) (q for quit):
```

We can see that `client` is waiting for us to specify a job string which should respect our format (see 4.1). We will give him the following job string : `20 1 3 /bin/ls -l`. The client terminal now displays the following content :

```
Enter your job string (--help for format & example) (q for quit):20 1 3 /bin/ls -l
You sent : 20 1 3 /bin/ls -l
Server responded : [+] Job added to queue
Enter your job string (--help for format & example) (q for quit):
```

We can see that the server responded that our job has been added to the Job queue. Everything is fine ! The client now wait for a new input.

Meanwhile, on our server/scheduler terminal we can see that our job is being processed :

```
[ ] Jobs updated by socket
[ ] Starting the job  /bin/ls -l time:1462209636
[ ] process pid 17750
total 2724
-rwxrwxr-x 1 archangel archangel  171736 mai    2 00:12 client
-rw-rw-r-- 1 archangel archangel   14395 avril 25 16:59 CMakeCache.txt
drwxrwxr-x 8 archangel archangel    4096 mai    2 13:57 CMakeFiles
-rw-rw-r-- 1 archangel archangel    1389 avril 25 16:59 cmake_install.cmake
-rw-rw-r-- 1 archangel archangel    6812 mai    2 13:57 compile_commands.json
-rw-rw-r-- 1 archangel archangel   19003 mai    2 13:57 Makefile
-rwxrwxr-x 1 archangel archangel  171736 mai    2 13:57 schedclt
-rwxrwxr-x 1 archangel archangel 1535048 mai    2 13:57 scheduler
-rwxrwxr-x 1 archangel archangel  787064 mai    2 00:12 tests
[ ] Stopping the job  /bin/ls -l (was running during 2 seconds)
```

Multiple client handling Our server/scheduler is capable of handling multiple client connections. You just have to launch multiple schedclt program.

7 Conclusion

Among all strategy, we preferred to focus our strategy on the user needs and experience as we empirically know that a good application is the one loved by its users.

References

[1] Wikipedia. Scheduling (computing), 2016. [Online; accessed 2-May-2016].