



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

GSI Project specifications

Auteurs : Titouan BION, Axel MARTIN et Thomas VIAU
Second year engineering students at the EISTI (Pau).

A l'attention de M. Juan ANGEL LORENZO DEL CASTILLO

March 21, 2016

Contents

1	Introduction	2
2	Features description	3
2.1	What will be implemented?	3
2.1.1	Job Queue management	3
2.1.2	Multiple versions	3
2.2	Gantt chart	3
3	Job Queue implementation details	4
3.1	Job parameters	4
3.2	Job Queue parameters	4
3.3	Dynamic feed of the Job Queue using <code>jobctl</code>	4
4	Scheduling strategy	5
4.1	Scheduling method description	5
4.2	Example of scheduling with our strategy	5
5	Conclusion	6

1 Introduction

As part of our Parallel Programming and Network courses, we are asked to develop a job scheduler. This document is the specifications of this project.

A scheduler is used to assign a job — processes, threads, etc. — to the machine hardware resources such as processors. All Operating Systems 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*".

First, we will present the features of our program. Then we will elaborate on the Queue management and finally we will develop on our chosen scheduling strategy.

2 Features description

2.1 What will be implemented?

We will implement a C++ Job Scheduler.

To begin with, let's clarify the definition of a job. A job will be a program launch by a command line. The latter will be given to our application — along with some parameters¹ — by being added to the Job Queue.

2.1.1 Job Queue management

A path to a file that contains all jobs information will be provide to our scheduler at start-up. All jobs included in this file will be processed by our scheduler.

However, we will provide a tool to interact with the scheduler to add new job on the fly². Jobs will be feed into the Job Queue regardless of the way it have been added and be processed by our program.

2.1.2 Multiple versions

First, we will develop a sequential management of our Job Queue, each job will be processed one by one and assigned to available resources according to their priorities³.

Then, we will provided a parallel version of our program to improve our Queue management and processing onto multiple cores of our processor with "OpenMP" library.

Finally, we will distribute the last version to a cluster of machine in order to make our program scalable with "MPI" library.

2.2 Gantt chart

[A FAIRE : *compléter diagramme de Fantt*] [A FAIRE : *Intégrer Gantt*]

¹See 3.1 for more information on job parameters.

²See our Job Queue implementation details for more information: 3.3

³See our Scheduling Method for more information: 4

3 Job Queue implementation details

3.1 Job parameters

A job will eventually have different properties — this list is not exhaustive and may differ at the end — provided by the user:

- Burst time [required]
- Priority
- RAM and CPU load

[\[A FAIRE : *explanations*\]](#)

3.2 Job Queue 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, by default there is no timeout and no job will be killed
- **Number of cores:** the number of processors the scheduler can send jobs to, by default it uses different values, depending on the version of the scheduler. In the sequential version it could not be modified and will be set to 1. In the parallel and distributed version it will match the number of available processors.
- **Time Slice:** if set to 0, there will be no time slice and the scheduler will be non-preemptive⁴.

3.3 Dynamic feed of the Job Queue using `jobctl`

A tool, `jobctl`, will be developed in order to interact with the scheduler. This tool will take as parameters the same information required to launch a job. It will then send a `SIGUSR1` signal to the scheduler, announcing the latter that it needs to reload its job queue. You can see an illustrated version of this routine on Figure 1. `jobctl` will also allow the user to see the current state of the processors — i.e. which job run on this processor.

The first implementation will use a swap file in order to add the jobs to the scheduler in real time. The swap file used for the job description and the communication between the client and the scheduler should not be edited by the user.

We may lately design a complete API in the scheduler daemon for retrieving or sending other informations such as the cores numbers in the cluster, parametrize the cluster or even remove process.

⁴See our Scheduling Method for more information: 4

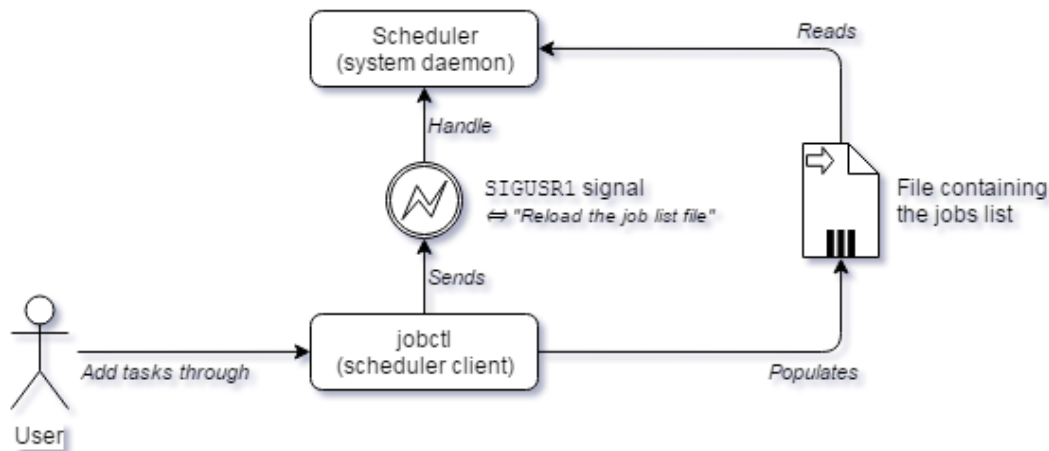


Figure 1: Chart of our Queue implementation.

4 Scheduling strategy

4.1 Scheduling method description

There are many methods and as many algorithms to achieve scheduling. We chose the fastest method but the most complex to imagine, this is the method used by the majority of Operating Systems, as Windows and UNIX systems (Mac, Linux ..).

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 jobs prioritized 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 as the time elapsed since job's declaration will raise the job's 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.

Moreover, the concept of 'time slice' is used in this method too. It means that a job will not hog the processor for a time longer than the 'time slice'. When a job is being runned by the scheduler, it can be suspended if a job with higher priority comes in, it will be resumed after the termination of the job with higher priority. Indeed, this 'priority race' is checked each 'time slice'. It can be turned off to become a Fixed-priority non-preemptive scheduling, but the pre-emptive method is the one we will use to ensure different jobs can be executed "simultaneously".

There will be 3 different versions of the application's scheduler, the 3 of them will be available through options. The 'Sequential', the most basic one, which only allow scheduling on 1 processor. The 'OpenMP', the multi-processors one, which allow using more than 1 processor to start jobs simultaneously. The 'MPI', the scheduler based on clusters, with many processors available on different machines.

4.2 Example of scheduling with our strategy

The following chart (Figure 2) 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

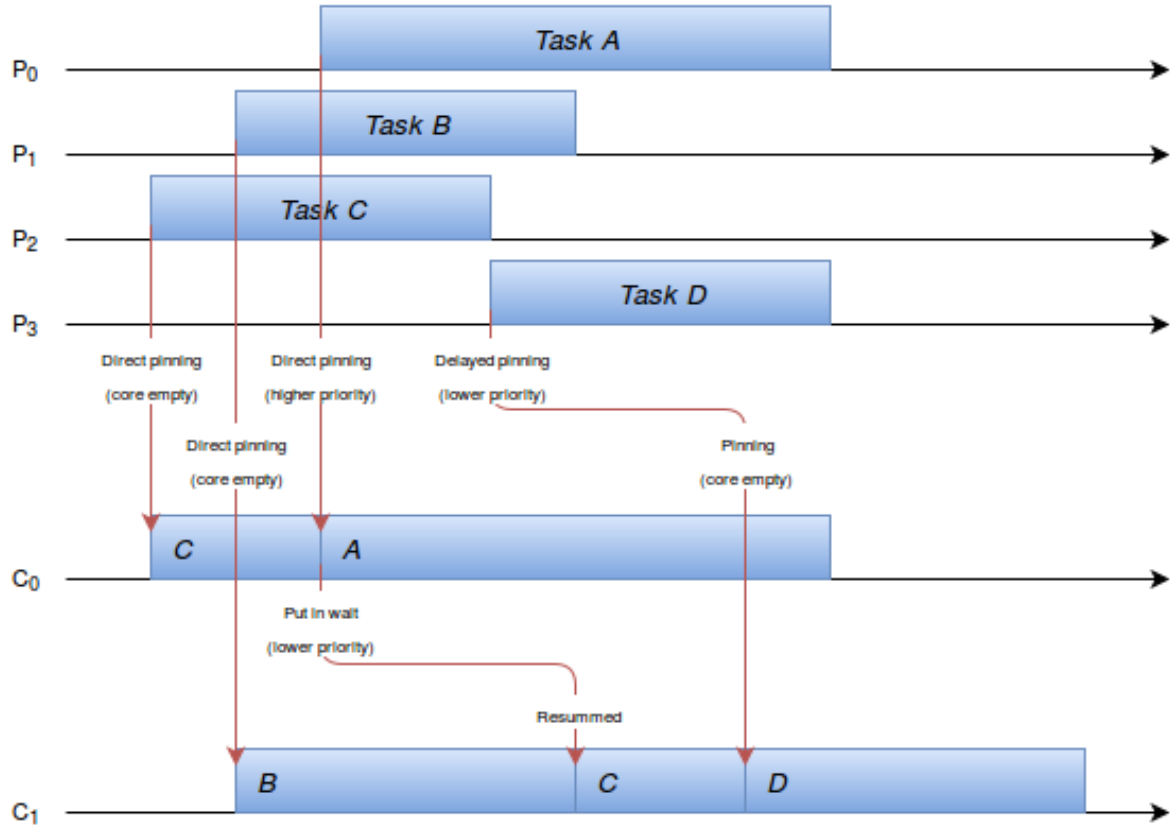


Figure 2: Example of scheduling following our strategy.

5 Conclusion

[A FAIRE : *formuler*]