



RAPPORT DE PROJET GSI

Ordonnancement dans des systèmes distribués

Étudiants :

Teddy GILBERT
Simon ESPIGOLÉ
Hugo LEGRAND

Responsable :

Juan Angel LORENZO DEL
CASTILLO

Pau, le 2 mai 2016

Remerciements

Nous voudrions remercier Juan Angel LORENZO DEL CASTILLO, pour son aide et ses conseils précieux lors des différentes séances de programmation ainsi que durant les cours qu'il nous a enseigné.

Aussi nous remercions Rémi VERNAY pour ses différents cours d'algorithmique et programmation parallèle qui nous ont permis de mieux comprendre ce projet et comment l'implémenter.

Table des matières

1	Remerciements	1
2	Introduction	4
2.1	Définition du sujet	4
2.2	Contexte et objectif	4
3	Présentation de l'interface utilisateur	6
4	Stratégie d'ordonnancement	11
4.1	Algorithme	11
4.2	Choix de la stratégie	11
4.3	Implémentation	12
4.3.1	En parallèle	12
4.4	Optimisation de l'ordonnancement	13
4.4.1	Tâches prioritaires	13
4.4.2	Expiration d'une tâche	13
4.4.3	Historique de session	14
5	Déroulement du programme	15
5.1	Lancement du programme et timeout	15
5.2	Remplissage de la file de message	15
5.3	Workflow	16
6	Problèmes rencontrés	17
6.1	Processus zombies	17
6.2	Portée des variables et accès concurrents	18

6.3	Parallélisation et échange de variables	19
6.4	Problèmes liés à l'horloge	19
7	Critique du module	20
7.1	Compréhension	20
7.2	Organisation du projet	20
8	Conclusion	21
9	Bibliographie	22

Introduction

2.1 Définition du sujet

L'ordonnanceur (**scheduler**, en anglais) est l'élément central d'un système d'exploitation multi-processus. L'ordonnancement (**Scheduling**) définit une méthode par laquelle un travail, **work**, va être attribué à un processeur pour une partie ou pour la totalité de son fonctionnement. Un ordinateur disposant de ressources limitées et un travail nécessitant des ressources pour fonctionner (puissance de calcul, capacité mémoire, etc...), l'ordonnanceur doit savoir répartir ces travaux afin d'optimiser les ressources matérielles à disposition du système. Il peut avoir plusieurs objectifs :

- Optimisation du nombre de travaux complétés par unité de temps (**Maximize throughput**)
- Minimiser le temps de réponse d'un processus en particulier (**Latency**)
- Minimiser le temps d'attente des programmes dans la file d'attente (**Responsiveness**)
- Égaliser les ressources disponibles pour chaque processus (**Fairness**)
- etc...

Il est très difficile d'implémenter un ordonnanceur capable d'optimiser plusieurs aspects à la fois. Par exemple, il faut faire un compromis entre le nombre de travaux complétés et le temps de réponse d'un processus en particulier (**Throughput VS Latency**). De plus, chaque optimisation de l'ordonnanceur correspond à une stratégie d'ordonnancement spécifique et peut avoir de sérieuses conséquences sur la stabilité et les performances du système d'exploitation.

2.2 Contexte et objectif

Dans le cadre du Projet GSI de seconde année d'ingénieur, nous avons réalisé un scheduler permettant d'ordonnancer différents processus selon une certaine stratégie d'ordonnancement. À réaliser en équipe de 3 personnes,

nous avons utilisé les méthodes vues en cours pour développer ce projet, qui nous a permis d'apprendre la programmation C++ et la programmation parallèle. De plus, pour la première fois notre outil de gestion de version de code (GitLAB) a été accessible par le responsable du projet, nous avons donc du adapter nos commits à cette situation, les décrire d'une façon plus claire, mais surtout utiliser une convention de nommage comme suit :

- Séparer le sujet du corps avec un saut de ligne
- Limiter le sujet à 50 caractères
- Mettre une majuscule au premier mot du sujet
- Ne pas finir le sujet par un point
- Utiliser l'impératif pour le sujet
- Limiter le corps à 72 caractères par ligne
- Utiliser le corps pour expliquer quoi, pourquoi et comment

Nous avons donc essayé d'aller au-delà des attentes classiques d'un projet Eistien pour avoir une approche encore plus professionnelle du projet informatique.

Présentation de l'interface utilisateur

Nous avons choisi une interface en ligne de commande pour ce projet. En effet, nous avons préféré nous investir dans le fond et terminer un maximum de fonctionnalités plutôt que de passer du temps sur une interface graphique dont l'intérêt pour un projet comme celui-là reste limité. L'interface présentée est celle de l'ordonnanceur séquentiel.

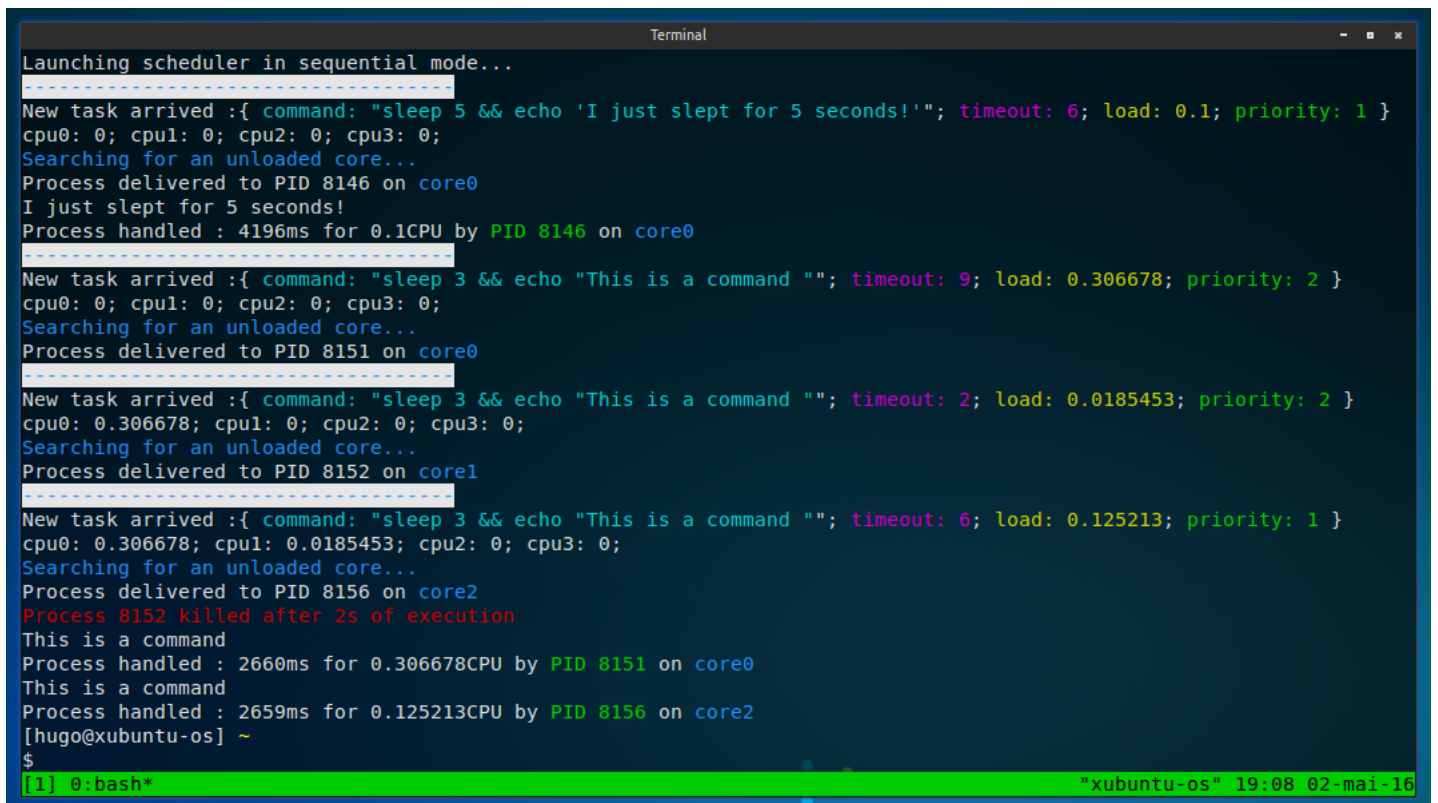
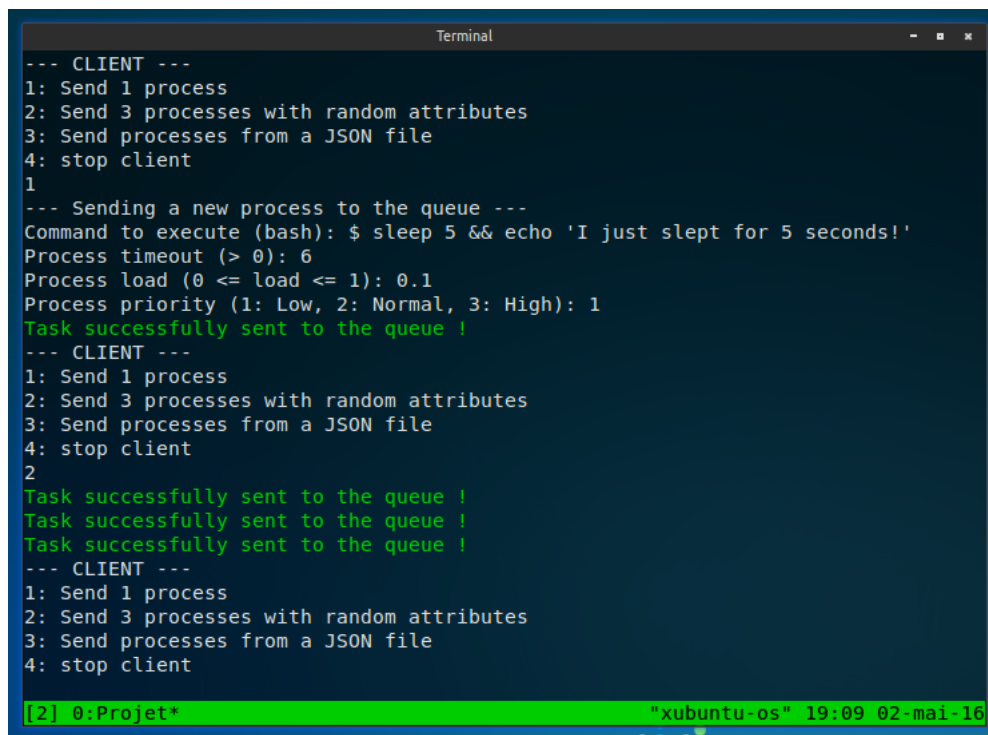
A screenshot of a terminal window titled "Terminal" with a dark blue background and light green text. The terminal displays the output of a scheduler program. It starts with "Launching scheduler in sequential mode...". Then, it shows three tasks arriving. Each task is a JSON object with fields: command, timeout, load, and priority. The first task has a 5-second sleep command, a 6-second timeout, a load of 0.1, and priority 1. It is delivered to PID 8146 on core0 and completes successfully. The second task has a 3-second sleep command, a 9-second timeout, a load of 0.306678, and priority 2. It is delivered to PID 8151 on core0 and completes successfully. The third task has a 3-second sleep command, a 6-second timeout, a load of 0.125213, and priority 1. It is delivered to PID 8156 on core2. After 2 seconds, it is killed. The terminal shows the command "This is a command" being executed twice. The terminal ends with a prompt "[1] 0:bash*" and a status bar at the bottom right showing "xubuntu-os" 19:08 02-mai-16.

FIGURE 3.1 – Affichage sheduler du programme



```
Terminal
--- CLIENT ---
1: Send 1 process
2: Send 3 processes with random attributes
3: Send processes from a JSON file
4: stop client
1
--- Sending a new process to the queue ---
Command to execute (bash): $ sleep 5 && echo 'I just slept for 5 seconds!'
Process timeout (> 0): 6
Process load (0 <= load <= 1): 0.1
Process priority (1: Low, 2: Normal, 3: High): 1
Task successfully sent to the queue !
--- CLIENT ---
1: Send 1 process
2: Send 3 processes with random attributes
3: Send processes from a JSON file
4: stop client
2
Task successfully sent to the queue !
Task successfully sent to the queue !
Task successfully sent to the queue !
--- CLIENT ---
1: Send 1 process
2: Send 3 processes with random attributes
3: Send processes from a JSON file
4: stop client
[2] 0:Projet* "xubuntu-os" 19:09 02-mai-16
```

FIGURE 3.2 – Affichage client du programme

Après avoir lancé le scheduler puis le client sur deux émulateurs de terminal différents, l'utilisateur interagit avec le client avec quatre menus principaux :

- Send 1 process
- Send 3 processes with random attributes
- Send processes from a JSON file
- Stop client

Il est intéressant de noter que l'ordre de lancement entre les deux programmes importe peu. En revanche, le fait de quitter le client n'arrête pas le scheduler. Lorsque le client est quitté, il faudra attendre que le scheduler s'arrête¹ (ou l'arrêter de manière brutale). En effet, le canal de communication est ouvert indifféremment par les deux programmes mais est fermé par le client. Cela est dû à une volonté de notre part de pouvoir manipuler le scheduler (démarrage, arrêt, coupure brutale) d'une façon très fluide.

Si l'utilisateur choisit la première option (Send 1 process), alors il lui

1. Voir 5.1 Lancement du programme et timeout

sera demandé plusieurs attributs à remplir pour que sa tâche soit envoyée au scheduler. Notamment :

- La commande à exécuter (en bash)
- Le temps d'expiration (timeout) de la commande (en secondes)
- La charge de cette commande sur un processeur (entre 0 et 1)
- La priorité de la tâche (entre 1 et 3)

Si la commande à exécuter n'est pas interprétable par un shell, le programme quittera. Après avoir rempli tous ces attributs, le client envoie la tâche au scheduler qui la traitera selon notre stratégie d'ordonnancement² puis imprime le message suivant à l'écran : "Task successfully sent to the queue!".

Si l'utilisateur choisit la seconde option (Send 3 processes with random attributes), le client va afficher les tâches qu'il va envoyer au scheduler : "created : command : "sleep 3 && echo "This is a command " "; timeout : 2; load : 0.627926; priority : 2 ", les envoie au scheduler puis affiche le message suivant : "Task successfully sent to the queue!".

Les résultats seront alors visibles sur l'écran du scheduler qui affiche son traitement comme suit, pour chaque tâche arrivée :

```
-----
New task arrived :{ command: "sleep 3 && echo "This is a command";
    timeout: 10; load: 0.960486; priority: 1}
cpu0: 0; cpu1: 0; cpu2: 0; cpu3: 0;
Searching for an unloaded core...
Process delivered to PID 10359 on core0
This is a command // Ici la commande est exécutée.
Process handled : 3010ms for 0.960486CPU by PID 10359 on core0
```

En outre, le client peut charger un fichier JSON à partir de la troisième option. Il lui faudra juste préciser le chemin ainsi que le nom du fichier à partir du dossier d'exécution du programme.

Si tout le fichier est écrit correctement alors les messages suivant s'afficheront après la demande du fichier. Si une tâche ne possède pas la bonne syntaxe, la lecture de fichier s'arrêtera et affichera une erreur.

2. Voir Chapitre 4 Stratégie d'ordonnancement

```

--- Enter the path of json file ---
test.json
Task successfully sent to the queue !
Task successfully sent to the queue !
....
Task cannot be created. Verify your json file.

```

Voici un template de fichier JSON que notre programme pourra analyser. Si jamais il manque un argument(command,timeout,load,priority) le programme entrera lui même l'argument manquant. Cela ne produira pas d'erreur et sera totalement transparent.

```

{
  "0" :{
    "command":"sleep 3 && echo \"This is a command\"",
    "timeout":10,
    "priority":3
  },
  "1":{
    "command":"sleep 3 && echo \"This is a command\"",
    "load":0.5,
    "priority":1
  },
  "2" :{
    "command":"sleep 3 && echo \"This is a command\"",
    "timeout":1,
    "load":0.7,
    "priority":2
  },
  "3" :{
    "command":"sleep 3 && echo \"This is a command\"
  },
  "4" :{
    "command":"sleep 3 && echo \"This is a command\"",
    "timeout":2,
    "load":0.98,
    "priority":3
  }
}

```

Le scheduler peut aussi afficher d'autres messages différents, mis dans le

désordre ici, qui seront détaillés plus tard :

```
...  
Process 10359 killed after 2s of execution // Voir 4.4.2 Expiration d'une tâche  
...  
No unloaded core found ! // Voir 4.1 Algorithme  
Searching for a core that can fit the task...  
...
```

Stratégie d'ordonnancement

4.1 Algorithme

Concernant l'algorithme d'ordonnancement utilisé, nous avons décidé de développer notre propre solution. Dans la section suivante, nous expliquerons pourquoi avoir fait ce choix. Voici la traduction en pseudo-code :

```
TANT QUE non timeout FAIRE
  -- Une nouvelle tâche arrive
  SI aucun processeur ne peut contenir la tâche
    Attendre qu'un processeur se libère
  SINON
    SI un processeur est vide (pas de tâches en cours)
      lui attribuer automatiquement la tâche.
    SINON
      Récupérer le processeur le moins utilisé
      Lui assigner la tâche
    FINSI
  FINSI
FIN TQ
```

4.2 Choix de la stratégie

Notre stratégie d'ordonnancement dérive de l'une des stratégies les plus répandues : **Load balancing** + FIFO. Cette stratégie implémente le principe d'une file d'attente. Le premier processus arrivé est affecté¹. Bien que relativement simple, cette solution a pour avantage de garder les ressources occupées et de ne pas surcharger un processeur grâce à la répartition des charges.

1. FIFO : First In First Out

Le développement de notre propre algorithme de `scheduling` nous a permis d'inclure certaines optimisations et fonctionnalités supplémentaires. Nous détaillerons ces fonctionnalités dans la section OPTIMISATION DE L'ORDONNANCEMENT.

Il est intéressant de noter que notre ordonnanceur vise en priorité les processeurs vides afin d'affecter une tâche. Nous avons donc une optimisation horizontale de la charge CPU. Cela diffère d'une optimisation verticale qui recherche à mettre le maximum de ressources sur un processeur avant de passer au suivant.

Raison : Il nous semble plus intéressant de chercher à paralléliser le plus possible les processus afin de gagner en temps d'exécution plutôt qu'à optimiser les ressources dans un système quasiment non contraint en CPU. De ce fait, nous utilisons toute la puissance disponible sur la machine.

4.3 Implémentation

Afin de pouvoir doter le programme d'une interface utilisateur fluide, nous avons opté pour une architecture **Client-Serveur**. Le client² permet de remplir la file de message de plusieurs manières³. L'utilisation de la file de message permet d'avoir un canal de communication quasi-instantanée entre le client et l'ordonnanceur tout en ne consommant que très peu de ressources (dû à son implémentation très bas niveau en C).

Le serveur⁴, séquentiel ou parallèle, lit tous les messages de la file de messages, selon leur ordre d'arrivée (FIFO).

4.3.1 En parallèle

Notre `scheduler` dispose d'un mode parallèle. Dans ce mode, certaines opérations d'ordonnancement ont été optimisées avec la librairie `OpenMP` afin de réduire le temps entre le moment où une tâche arrive et le temps où elle est affectée à un processeur.

Les fonctions parallélisées sont les suivantes :

2. Le client s'exécute avec la commande `Scheduler -c` (ou `-client`)

3. Cf. PRÉSENTATION DE L'INTERFACE UTILISATEUR

4. Le serveur s'exécute avec la commande `Scheduler -s` pour le mode séquentiel ou `Scheduler -p` pour le mode parallèle

- Recherche d'un processeur capable de contenir la tâche
- Recherche d'un processeur vide
- Recherche du processeur le moins chargé

Il est important de noter que la file de message de l'ordonnanceur reste la même quelque soit le mode. Il est donc tout à fait possible de traiter quelques tâches en mode séquentiel, d'arrêter le programme et de le relancer en mode parallèle afin de terminer les affectations.

Malgré nos efforts d'implémentation, il ne s'agit pas réellement d'un **scheduler** parallèle. En effet, les tâches ne sont pas affectées à un processeur de manière simultanée mais seulement quelques opérations de recherche ont été optimisées.

4.4 Optimisation de l'ordonnancement

Comme précisé précédemment, le développement de notre propre algorithme d'ordonnancement nous a permis d'y inclure quelques fonctionnalités supplémentaires. Ces améliorations sont disponibles en mode séquentiel et parallèle.

4.4.1 Tâches prioritaires

Lorsque l'utilisateur crée une tâche, il doit lui affecter une priorité⁵. Cette priorité permet à l'ordonnanceur de savoir que la tâche doit être affecté le plus rapidement possible. Ainsi, dans la file de message, le **scheduler** cherchera toujours à assigner la plus ancienne tâche (arrivée la première) avec la priorité la plus élevée.

4.4.2 Expiration d'une tâche

Nous avons implémenté un système permettant de tuer une tâche trop longue à s'exécuter. En effet, si le temps d'exécution **réel** dépasse le **timeout**⁶, la tâche expire et est tuée par l'ordonnanceur.

Cette méthode n'est pas une étape en plus dans la boucle⁷ pour le **scheduler** car elle ne s'exécute pas dans le **thread** principale du programme. En effet, elle est encapsulée dans l'exécution de la tâche elle-même, ce qui permet la libération de charges sur le CPU tout en continuant l'affectation.

5. Cf. PRÉSENTATION DE L'INTERFACE UTILISATEUR

6. Cf. PRÉSENTATION DE L'INTERFACE UTILISATEUR

7. Cf. ALGORITHME

Pour schématiser, son comportement est semblable à une tâche qui détecterait elle-même qu'elle dépasse le temps que le processeur lui avait accordé et se "suiciderait". Cette fonctionnalité, d'une extrême simplicité, est très efficace et permet de simuler l'ordonnancement dans un système contraint en temps.

4.4.3 Historique de session

Bien que ce ne soit pas une optimisation d'ordonnancement (car cela n'améliore pas les performances du `scheduler`), le programme dispose d'un journal contenant l'historique de sa dernière utilisation. Ceci permet d'analyser le comportement de l'ordonnanceur sans avoir à garder un terminal ouvert. Le journal se situe à l'endroit depuis lequel le programme est exécuté.

Déroulement du programme

5.1 Lancement du programme et timeout

Au lancement du programme scheduler, un temps d'expiration (timeout) est lancé. Passé ce timeout, le programme quittera automatiquement. Le timeout est de 60 secondes au lancement du programme, et est relancé à chaque tâche reçue par le scheduler pour la valeur du timeout de la tâche entrée par l'utilisateur auquel nous additionnons 60 secondes. L'utilisateur a donc en moyenne une minute pour interagir avec le programme avant que celui-ci ne quitte. Si le programme scheduler quitte, le relancer suffira pour de nouveau interagir avec le programme client, même si celui-ci n'a pas été fermé.

5.2 Remplissage de la file de message

Afin que l'ordonnanceur puisse travailler, il faut tout d'abord remplir la file avec des tâches à consommer. Pour cela, exécuter le programme en mode client :

```
$ ./Scheduler --client
```

Puis choisissez le mode d'envoi des messages que vous souhaitez. Vous pouvez laisser le client ouvert et continuer à envoyer des tâches dans la file pendant l'exécution de l'ordonnanceur. Ne pas oublier d'ouvrir un scheduler :

```
$ ./Scheduler --sequential  
$ ./Scheduler --parallel
```


5.3 Workflow

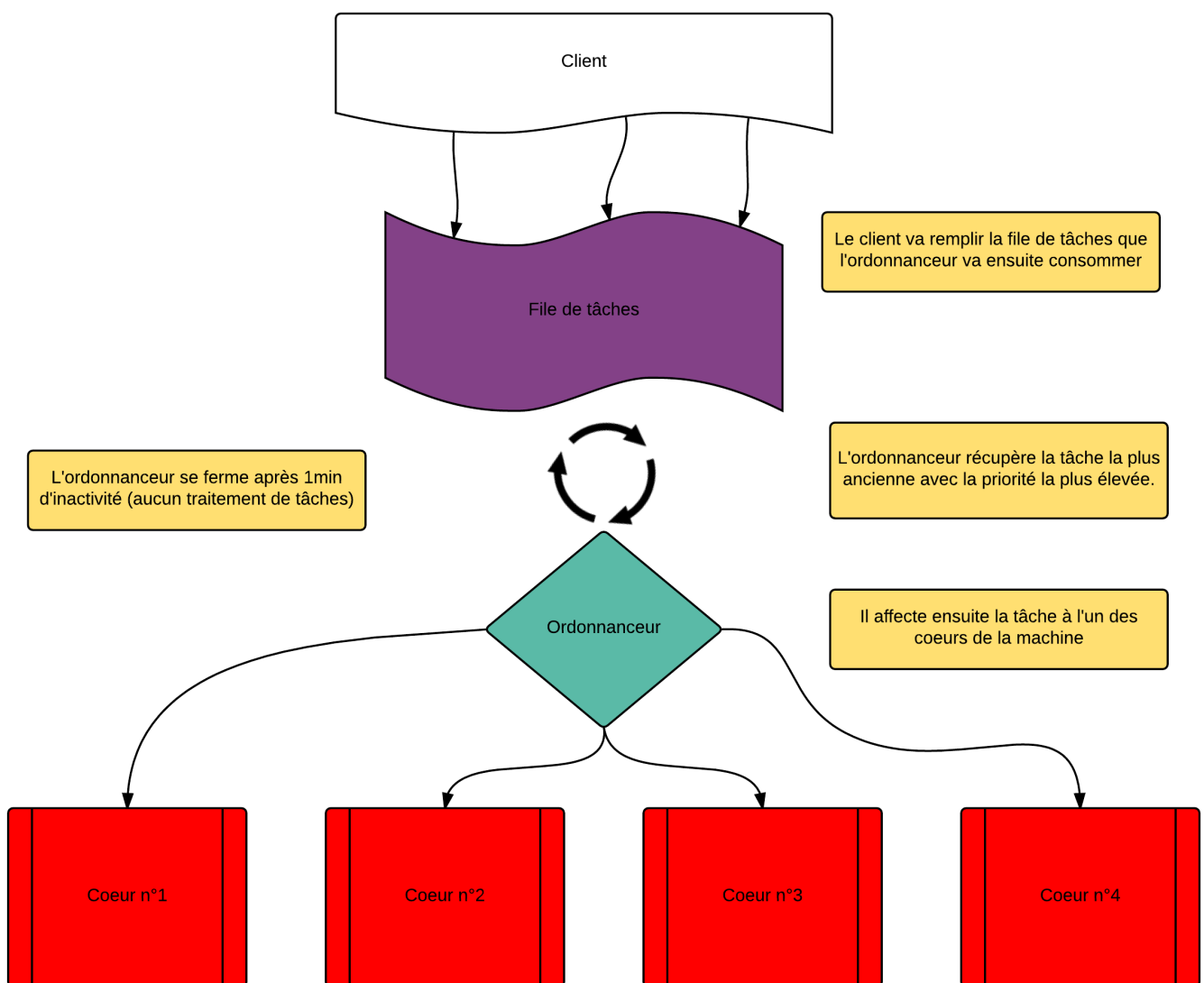


FIGURE 5.1 – Workflow (happy path) de notre programme

Problèmes rencontrés

Pendant le développement de ce projet, nous avons chacun rencontré différents problèmes. Heureusement composé de trois membres, notre groupe a su répondre aux problématiques de chacun de ses membres en s'entraidant dans les difficultés rencontrées par tous les protagonistes. En outre, cette entraide nous a permis de mieux cerner le sujet mais aussi mieux comprendre les enjeux de la parallélisation.

6.1 Processus zombies

Du fait de la parallélisation, nous avons connu quelques problèmes avec les processus zombies. En effet, nous lançons la commande demandée par l'utilisateur sur un processus nouveau, ce qui nous permet de la contrôler entièrement (finir son exécution ou la tuer). Or, certains processus restaient en mode zombie plutôt que de se terminer proprement. Après quelques recherches, nous avons trouvé la solution en faisant comme suit :

```

if (exec == 0) {
    setpgid(getpid(), getpid());
    std::system(command.c_str());
    ...
    exit(EXIT_SUCCESS);
} else {
    sleep(_task.timeout);
    pid_t result = waitpid(exec, &status, WNOHANG);
    // If his child is still alive...
    if (result == 0) {
        // Kill the process because the timeout is exceeded
        kill(-exec, SIGTERM);
        ...
    }
}

```

En effet, pour terminer un fils proprement, il faut envoyer le signal `EXIT_SUCCESS`. Le père, si le timeout est dépassé, va récupérer le signal de fin d'exécution du fils avec la commande `WAITPID`. Si le résultat vaut 0 (signal renvoyé si le fils est toujours en vie), alors le père va décider de le tuer avec la commande `KILL(-EXEC, SIGTERM)`. Il faut noter que le "-" placé devant le PID "exec" permet de rendre le PID négatif et donc de tuer tous les processus associés au groupe de processus ayant pour PGID "exec". Le PGID est défini grâce à la ligne `SETPGID(GETPID(), GETPID())` ;.

6.2 Portée des variables et accès concurrents

L'une des premières difficultés lors de l'exécution parallèle des tâches a été la portée des variables du programme. En effet, lors d'un `fork`, les variables communes entre le processus père et son fils ne sont pas partagées mais copiées. Précisément, un fils a accès aux variables de son père mais leur contexte reste celui juste avant le `fork`. De ce fait, le père ne peut en aucun cas avoir connaissance des changements apportés par son fils sur ces variables. Pour y faire face, nous les avons placées à l'intérieur de conteneur, des **Shared Memory**. En procédant de cette manière, plusieurs processus peuvent lire et modifier une même variable à travers un programme.

Cependant, la lecture et l'écriture simultanée d'une variable, d'un fichier texte ou d'une structure entraîne des accès concurrents. Nous avons dû sécuriser ces opérations grâce à des `mutex`. Il s'agit d'un système de jeton permettant de restreindre l'accès d'un objet à un seul et unique processus à

un instant t .

6.3 Parallélisation et échange de variables

Lors de la parallélisation du scheduler, nous avons rencontré plusieurs problèmes de concurrences. Tout d'abord, en appliquant une simple parallélisation des boucles for pour la recherche du CPU. Un problème est alors survenu sur la charge des CPU. Ils dépassaient leur charge possible (>1). Il a fallu partager avec un `#PRAGMA OMP SHARED` les boucles afin d'éviter ce problème.

De plus, lors de la recherche d'un CPU libre, la boucle for se coupait si l'un des CPU était libre. Or en parallélisation, cela n'est pas possible. Donc nous avons rajouté une variable de stockage. Puis à la fin de la parallélisation, nous avons fait une somme logique grâce au `#PRAGMA OMP REDUCTION`.

Pour finir, un problème venait de la parallélisation des affectations. Cela n'est pas possible hormis si nous mettons un endroit critique afin que tous les threads n'affectent pas leur valeur en même temps. Cette fonction est donc restée non parallèle. Ce sont les limites de la parallélisation.

6.4 Problèmes liés à l'horloge

Lors de l'implémentation d'une horloge pour mesurer le temps d'exécution réel d'une commande passée par l'utilisateur, nous avons remarqué que pour une commande censée durer 5 secondes, l'horloge nous affiche un temps de 4100ms à 4900ms environ. En effet nous pensons que cela est dû au temps d'initialisation de l'horloge, qui varie selon l'exécution. Nous n'avons cependant pas trouvé de solution à cette problématique.

Critique du module

7.1 Compréhension

Le projet de simuler un `scheduler` est très intéressant. Cela nous a permis de comprendre un peu plus comment fonctionnait l'ordonnancement des tâches sur un ordinateur. Le sujet a été très clair concernant les technologies à utiliser et les modalités de livraison. Cependant, nous regrettons un certain flou par rapport à la limite qui différenciait l'ordonnanceur séquentiel de l'ordonnanceur parallèle.

7.2 Organisation du projet

L'organisation des groupes était très bien car nous avions le choix d'être avec les personnes que nous voulions. Il est vrai que des groupes imposés aurait fortement handicapé et ralenti un projet dont les délais étaient déjà assez court.

Un des problèmes était qu'au début du projet nous ne connaissions rien et nous avons fait les cours en parallèle. Nous ne pensons pas que cela soit une bonne stratégie. Certes, les compétences sont mises directement en application mais le projet avance moins vite et c'est un peu plus brouillon. Le fait d'avoir les connaissances au début du projet nous aurait permis de ne pas faire d'erreur au départ et d'avoir un planning à long terme. Il a fallu plusieurs fois modifier le code car il ne correspondait plus à ce que nous pouvions ni devions faire.

Conclusion

Ce projet a été pour nous une source d'apprentissage inconsiderable. Ce dernier nous a permis de mieux comprendre une grande partie des matieres enseignees au second semestre. De plus, il a été un support pour ces matieres tout au long de cette periode. Il a été parfois difficile de rester en phase avec les cours puisque nous avons besoin de plus de materiel pour continuer à travailler sur le projet.

Bien qu'il fut compliqué de comprendre au depart les subtilites de ce projet, nous avons réussi en un temps raisonnable à nous approprier ce dernier et commencer à faire les premieres fonctionnalites.

Nous avons, selon nous, fait un travail de qualite sur ce projet en nous concentrant sur un code propre, bien écrit, commenté, documenté (grâce à doxygen), et respectant un maximum de normes de bon sens. De plus, nous nous sommes efforcés de faire des commits utiles sur Git, pour une comprehension accrue de notre repository.

Pour toutes ses raisons, nous pensons avoir réussi notre projet sur la forme en premier lieu puisque nous avons fait de notre mieux pour nous engager dans une demarche serieuse et professionnelle. Bien sûr, nous avons aussi appris de ce projet sur cette demarche et savons ce que nous devons refaire et ce que nous ne devons pas

En conclusion, nous avons tenté de mêler technique et gestion de projet, en utilisant un maximum de connaissances qui nous ont été enseignees durant nos differentes annees d'etudes à l'EISTI, et sommes fiers du resultat obtenu. De plus, nous avons trouvé ce sujet très interessant puisque techniquement élevé et aimerions retravailler sur un projet de ce type à l'avenir.

Bibliographie

- <http://bisqwit.iki.fi/story/howto/openmp/>
- http://www.boost.org/doc/libs/1_60_0/
- http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html

Table des figures

3.1	Affichage sheduler du programme	6
3.2	Affichage client du programme	7
5.1	Workflow (happy path) de notre programme	16