Cahier des charges

Ordonnancement dans des systèmes distribués

Étudiants :
Teddy Gilbert
Simon Espigolé
Hugo Legrand

Responsable :
Juan Angel LORENZO DEL CASTILLO

Pau, le 21 mars 2016

Table des matières

Chapitre 1

Introduction

1.1 Définition du sujet

L'ordonnanceur (`scheduler`, en anglais) est l'élément central d'un système d'exploitation multi-processus. L'ordonnancement (`Scheduling`) définie 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 matériels 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.

1.2 Contexte et objectif

Dans le cadre du Projet GSI de seconde année d'ingénieur, nous réaliserons un scheduler permettant d'ordonnancer différents processus selon certaines stratégies d'ordonnancement que nous définirons. À réaliser en équipe de 3 personnes, nous utiliserons les méthodes vues en cours pour développer ce projet. Ce projet a pour but de nous faire apprendre la programmation C++ et la programmation parallèle ainsi que distribuée, choses que nous nous efforcerons au mieux de réaliser et de comprendre. De plus, pour la première fois notre outil de gestion de version de code (GitLAB) sera accessible par le responsable du projet, nous devrons donc adapter nos commit 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 essaierons donc d'aller au-delà des attentes classiques d'un projet Eistien pour avoir une approche encore plus professionnelle du projet informatique.

Chapitre 2

Cahier des charges

Les spécifications fournies par le client sont claires : le but de ce projet est d'implémenter un ordonnanceur, un programme permettant d'affecter des tâches à un processeur en fonction de plusieurs options ou paramètres comme :

— Temps d'exécution nécessaire
— Puissance de calcul demandé (utilisation du processeur)
— Allocation mémoire demandée

Dans un premier temps nous nous limiterons à une seule stratégie d'ordonnancement : `Load balancing` + FIFO.
Cette stratégie implémente le principe d'une file d'attente. Le premier processus arrivant sera affecté en premier [1].
Bien que simpliste, cela permettra de garder les ressources occupées et de ne pas surcharger un processeur grâce à la répartition des charges.
Néanmoins, nous prévoyons plus tard de permettre à l'utilisateur de choisir sa stratégie d'ordonnancement via une interface terminale simple et intuitive.
Afin de répartir les charges sur les différents coeurs du processeur, nous utiliserons une option associée au travail que l'on souhaite exécuter. Cela pourrait être, par exemple, une fraction du CPU à utiliser.

Les tâches à effectuer devront être décrites dans un fichier texte. Le fichier pourra avoir la structure suivante :

```
COMMAND, [OPTIONS]
--------
echo "This is a task", 0.1CPU
ls -l ~/*, 0.2CPU
rm -rf /, 1CPU
```

1. FIFO : `First In First Out`

Ce fichier texte sera analysé et les travaux à effectuer seront placés dans une liste d'attente [2] dans le programme. Puis, l'ordonnanceur va créer un processus pour chaque commande dans la liste d'attente et les affecter à un `core` suivant la stratégie d'ordonnancement que l'on aura définie en amont. Le programme se termine lorsque la file d'attente ne contient plus aucune tâche à traiter.

La réalisation de ce `scheduler` se fera sous différentes formes :

— Séquentiel
— Parallèle Mono-machine
— Parallèle Multi-machine

La première version de l'ordonnanceur sera la plus basique. En effet, les travaux seront assignés les uns après les autres vers le processeur.
La deuxième implémentation est nettement plus intéressante : le traitement des tâches de la liste se fera en parallèle afin que plusieurs travaux puissent être assigné simultanément. Il faudra donc porter attention aux problèmes de concurrence et de synchronisation qu'il va en résulter.
Finalement, il s'agira de répartir les tâches entre différentes machines sur un réseau. Nous détaillerons plus précisément la démarche que nous mettrons en oeuvre dans la partie `Implémentation` de ce cahier des charges.

2. `Task queue` ou `pool`, en anglais

Chapitre 3

Implémentation

Pour implémenter ce scheduler, nous utiliserons le C++, avec les librairies OpenMP et MPI. Les connections réseau seront faites en IPv6, et le code source sera partagé sur le GitLAB de l'École. Notre groupe utilisera l'environnement de développement CLion (JetBrains) pour toute la rédaction du code source, ainsi que le partage. Pour chaque thread (un par processeur), nous irons chercher une tâche dans le pool de tâches que nous aurons généré au préalable, puis nous la supprimerons. Nous exécuterons la tâche, puis attendrons jusqu'à ce que une nouvelle tâche soit disponible. Le programme s'arrête lorsqu'il n'y a plus de tâche dans le pool de tâches.

3.1 Partie séquentielle

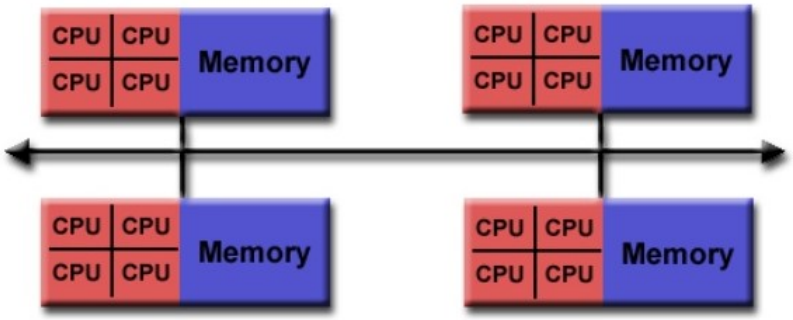
Pour la partie séquentielle, nous avons pour idée de créer une liste de tâche. Cette liste sera récupérer après un traitement d'analyse de fichier texte afin d'avoir l'ensemble des options d'une tâche. Une fois ceci fait, chaque tâche sera traité l'une après l'autre. Une tâche devra être affectée à un CPU disponible et étant capable d'accueillir la tâche afin de la traiter. Après son affectation, une nouvelle tâche sera récupérée dans la liste. Le programme se terminera quand l'ensemble de la liste sera vide.

3.2 Partie parallèle mono-machine

Pour la partie parallèle mono-machine, nous allons repartir de la version séquentielle en y ajoutant diverses options. Le traitement des tâches se fera en même temps, une tâche aura un certain poids et se verra affecter un CPU libre et pouvant l'accueillir. Le traitement des tâches se fera de façon parallèle afin de traiter plus rapidement la liste de tâches

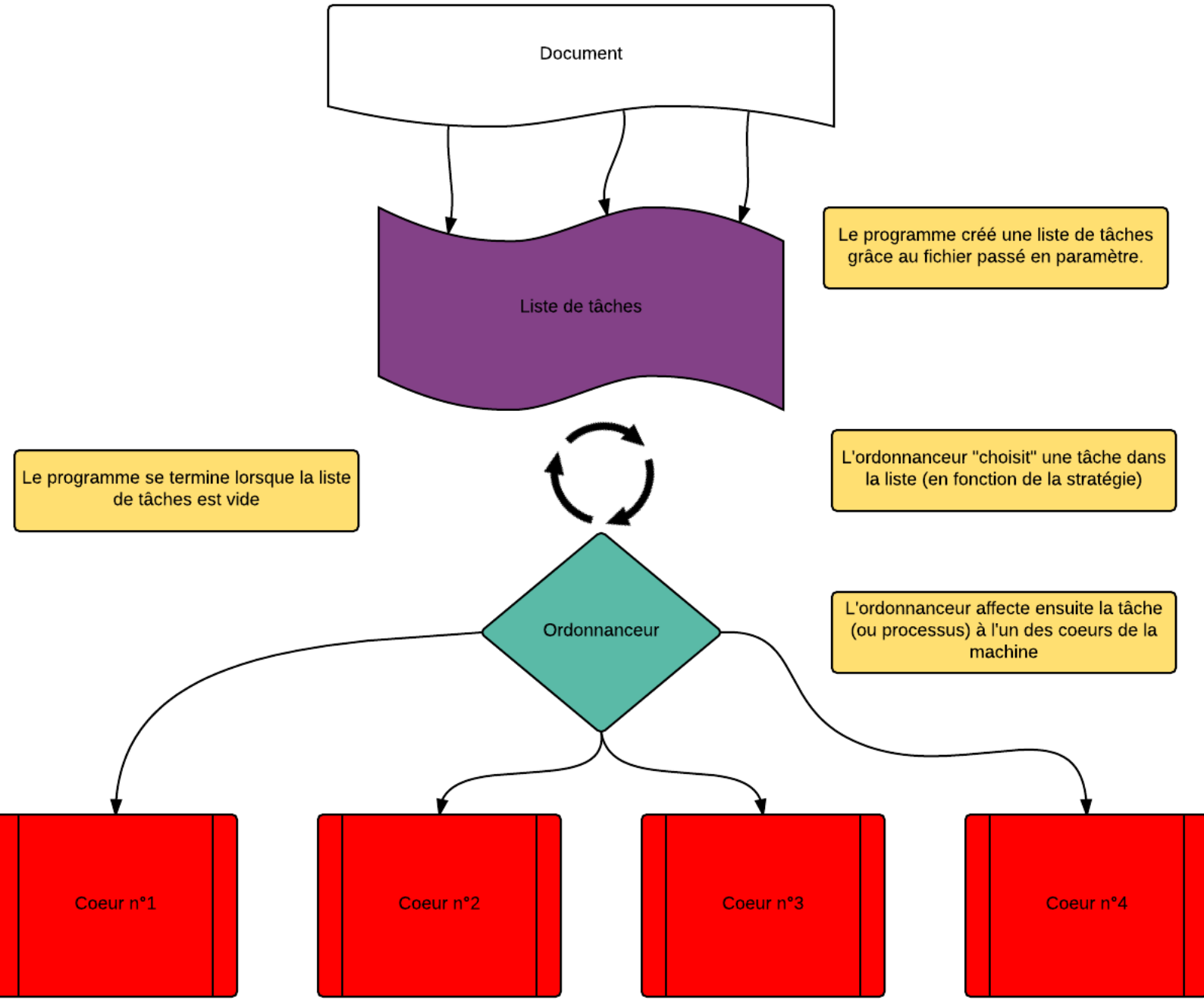
3.3 Partie parallèle multi-machine

La version parallèle de l'ordonnanceur sera réalisée grâce à la librairie MPI en C++. De ce fait, nous serons en mesure d'envoyer un processus directement sur le processeur d'un ordinateur sur le même réseau.



3.4 Workflow - Happy path

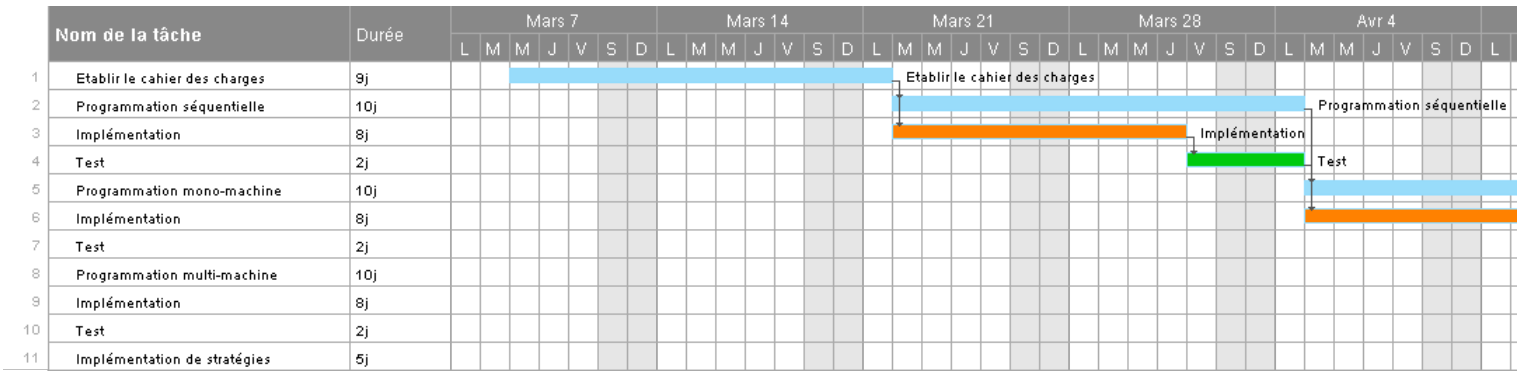
Ci-dessous, vous trouverez le chemin idéal que l'application prendra lors de l'exécution normale du programme :

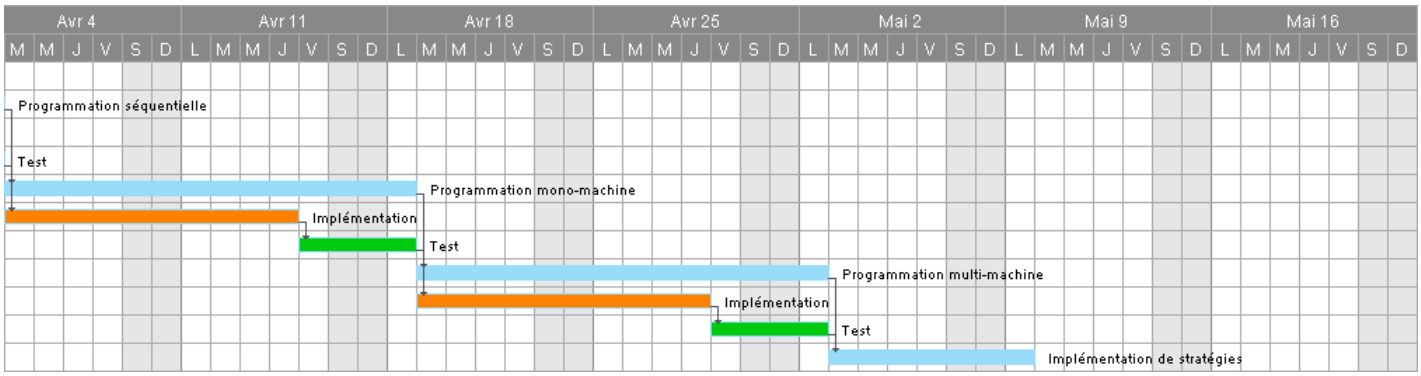


Chapitre 4

Planning prévisionnel

Nous avons prévu un planning pour l'instant simple : nous y avons intégré les grandes étapes du développement sans oublier les phases de test et d'implémentation.





Chapitre 5

Axes d'amélioration

5.1 Amélioration d'interface

S'il nous reste du temps après avoir implémenté les principales fonctionnalités, nous choisirons de développer une interface de commande un peu plus développée en premier lieu. Nous souhaitons à terme pouvoir proposer des options nombreuses pour paramétrer au mieux le scheduler.

5.2 Autres stratégies d'ordonnancement

5.2.1 Fixed priority pre-emptive scheduling

Une priorité est fixé aux processus. Le scheduler ordonne la file d'attente en fonction de cette priorité.

— Revient à faire plusieurs FIFO si le range de priorité est limité.
— Accorde une réponse quasi immédiate au processus de plus haute priorité

5.2.2 Round-robin scheduling

Un jeton [1] est passé entre les tâches. La tâche qui possède le jeton va s'exécuter pendant un certain temps défini par l'utilisateur, puis l'ordonnanceur va passer le jeton à la tâche suivante. Une fois les processus tous complétés, on arrête.

Cette implémentation risque d'être difficile car il faut sauvegarder le contexte d'un processus avant de le mettre en pause et de passer à un autre

1. `Token`, en anglais

processus. Il sera donc intéressant de vérifier les performances de cette stratégies selon les différents temps définis par l'utilisateur.

Nous pourrons par la suite gérer un peu mieux les ressources en instaurant un timeout pour toutes les tâches gérées, qui seront tuées si elles dépassent ce laps de temps. Puis une hiérarchisation des tâches sera effectuée : des priorités seront assignées aux tâches et les plus importantes seront exécutées en premier.

Chapitre 6

Conclusion

Ce projet nous servira de base pour apprendre à programmer en C++. Nous nous efforcerons de suivre une méthode rigoureuse tout au long du projet, tant dans la communication inter et intra groupe que dans notre façon de coder. De plus, la perspective du développement parallèle et distribué nous intéresse : c'est en effet une nouvelle manière de penser que nous admirons et dont nous avons beaucoup à apprendre. Nous essaierons alors de réaliser au mieux les objectifs que nous nous sommes fixés.

Pour finir, il nous semble important de prendre un peu de recul sur le projet. Bien que nous nous efforcerons d'implémenter la meilleure version de `scheduler` possible, il semble prétentieux de penser que celui-ci serait viable, au sein d'un système d'exploitation par exemple. En effet, la grande majorité des ordonnanceurs utilisés dans les systèmes informatiques grand public ont des stratégies très différentes de celle que nous programmerons. Bien souvent, leur stratégie consiste en une fusion de plusieurs stratégies connues et optimisées.