Chapitre 13 : Concurrence et synchronisation

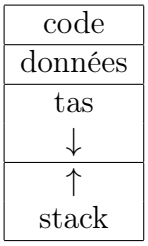
Table des matières

1 Introduction

1.1 Motivation

1.1.1 Rappel (chap.2, 1.1.1)

Un programme en cours d'exécution dispose d'un espace mémoire dédié organisé comme suit :



Le tas (*heap*) est la zone mémoire qui contient les données allouées dynamiquement.

La pile (*stack*) contient toutes les données liées à la gestion des appels de fonction.

Dans ce chapitre, un programme en cours d'exécution sera appelé *processus* et le terme *programme* fera référence au code.

1.1.2 Concurrence

En pratique dans un ordinateur, il y a plusieurs processus actifs simultanément, qui doivent se partager les ressources de la machine (mémoire, entrées/sorties, unité de calcul).

Le programme de MPI se limite à l'étude de machines ayant une unique unité de calcul. En particulier, cela signifie qu'il ne peut pas y avoir plusieurs processus actifs en même temps.

Pour contourner ce problème, le système met es place une alternance de processus : on exécute quelques instructions d'un processus avant de changer de contexte pour exécuter un autre processus. Si les changements de contexte sont assez rapides, l'utilisateur a l'impression d'une vraie exécution parallèle.

Problème : les changements de contexte sont lents. De plus, deux processus ne sont pas forcément indépendants.

Exemple : Louis tape ses cours en LaTeX, et doit exécuter un compilateur pour obtenir un document PDF. Il peut alors le visionner à l'aide d'un autre programme qui lit dans la même zone mémoire que celle où le compilateur écrit. À chaque mise à jour, le programme de lecture doit rafraîchir l'affichage.

1.1.3 Non déterminisme et synchronisation

L'exécution de processus concurrents est non déterministe car on ne peut pas faire l'hypothèse sur l'ordre d'exécution des instructions et des changements de contextes. En effet, le système d'exploitation, *via* un programme appelé *ordonnanceur*, décide es

changements de contexte selon des critères variés (horloge, événement provoqués par l'utilisateur, attente de données qui proviennent de la mémoire, ...)

Ce non-déterminisme implique la nécessité de synchroniser certains processus.

Par exemple, on considère deux processus concurrents qui exécutent le même programme.

Le code est le suivant :

> Répéter 100 fois :
> Lire l'entier n dans le fichier "`toto.txt`"
> Écraser le fichier "`toto.txt`" en y écrivant $n+1$

On suppose qu'initialement, le fichier contient l'entier 0. Quel est le résultat final ?

C'est une valeur de $[\![100\ ;\ 200]\!]$ car un processus peut être interrompu juste après une lecture et l'écriture qui suivra la reprise de son exécution écrasera tous les changements faits par le second processus).

1.2 Définition et objectifs

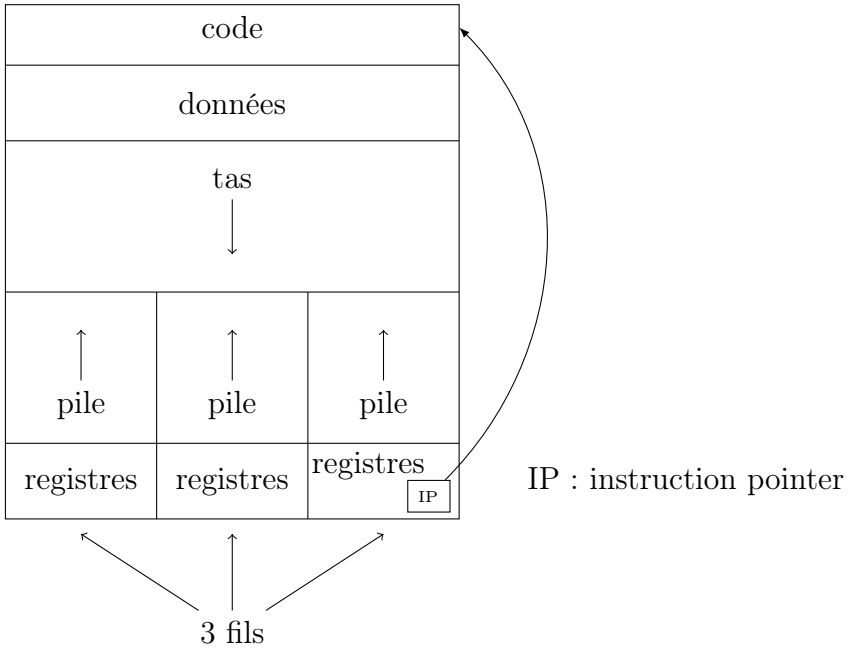
1.2.1 Fil d'exécution (*thread*)

Dans certains cas, un processus peut être amené à effectuer plusieurs tâches, que l'on pourrait vouloir répartir sur plusieurs processus. Si les tâches ne sont pas indépendantes, on peut être amené à faire de nombreux changements de contexte qui peuvent être coûteux.

Il faut de plus un moyen de communication entre ces processus, qui peut être une interruption système (le système d'exploitation sert de messager) ou un partage de mémoire.

En réalité, un processus peut contenir plusieurs fils d'exécution ou processus légers qui partagent une partie de la mémoire du processus. Ces fils d'exécution ont un programme commun mais l'exécutent en des points différents.

La structure de la mémoire d'un processus devient la suivante :



Limitations du programme :

- On étudie les fils d'exécution d'un unique processus ;
- On se limite uniquement à la norme POSIX ;
- On s'autorise uniquement deux primitives sur les fils d'exécutions :
  - `create` : prend une fonction et des paramètres pour cette fonction et crée un nouveau fil d'exécution pour le processus courant, chargé d'évaluer la fonction sur les paramètres ;
  - `join` : prend l'identifiant d'un file d'exécution en paramètre et interrompt le fil courant jusqu'à terminaison du fil passé en argument.

1.2.2 Situation de compétition et atomicité

Un *situation de compétition* (*race condition*) a lieu quand le résultat d'un processus varie selon l'ordre d'exécution de ses fils. L'exemple vu en 1.1.3 (page 2) est une situation de compétition que l'on eput reproduire avec les fils d'un unique processus partageant une variable globale. Un fil principal est chargé de créer deux fils secondaires qui exécutent la boucle. Il doit attendre la terminaison des fils secondaires (opération `join`) avant de s'arrêter pour éviter une interruption prématurée du programme.

Dans un cas comme celui-ci, on peut résoudre le problème en imposant *l'atomicité* du corps de la boucle. Un ensemble d'instructions est *atomique* si le système ne peut pas interrompre leur exécution. Ici, on ne voit pas d'interruption entre la lecture et l'écriture.

1.2.3 Section critique

Dans l'exemple précédent, l'atomicité est une condition trop forte : on peut interrompre un fil entre la lecture et l'écriture tant que l'autre fil ne lit et n'écrit pas dans la variable.

On dit que le corps de la boucle est une *section critique* du programme.

On veut garantir l'exclusion mutuelle pour les secteurs critiques, *i.e* il ne peut y avoir qu'un seul fil qui exécute une section critique à chaque instant.

Une solution de synchronisation doit aussi garantir le progrès de l'exécution : si un fil souhaite entrer en section critique et si celle-ci est libre, le choix d'un fil entrant en section critique ne doit pas pouvoir être retardé indéfiniment.

On veut aussi assurer un temps d'attente borné : il doit y avoir une borne sur le nombre de fois que d'autres fils d'exécution entrent en section critique entre le moment où un fil signale qu'il souhaite entrer en section critique et son entrée effective.

1.2.4 Interblocage (*deadlock*) et famine

Si la synchronisation des fils d'exécution n'est pas effectuée correctement, il peut y avoir plusieurs problèmes, dont :

• l'interblocage : on dit qu'il y a *interblocage* lorsque plusieurs fils attendent un événement qui ne peut être provoqué que par l'un des fils en attente.

Exemple classique : le dîner des philosophes.

Des philosophes sont réunis autour d'une table ronde, et on deux activités : manger et penser. Pour manger, un philosophe doit disposer de deux baguettes. Les baguettes sont réparties comme suit : il y en a une entre chaque couple de philosophe voisins.

Si les philosophes exécutent le programme suivant :

```
Prendre la baguette à gauche
Prendre la baguette à droite
Manger
Poser les baguettes
Penser
```

Il peut y avoir unterblocage si chaque philosophe a pris la baguette à sa gauche et attend que son voisin de droite libère l'autre.

• La *famine* a lieu lorsqu'un fil d'exécution attend indéfiniment l'accès à une ressource.

Exemple classique : le problème des producteurs-consommateurs.

Des producteurs remplissent un buffer que les consommateurs vident.

Règles : une donnée ne peut être lue qu'une fois, on ne peut pas écraser une donnée qui n'a pas été lue, et une case vide ne peut pas être lue.

Si l'accès au buffer n'est pas équitable (par exemple une priorité selon les identifiants des fils d'exécution), un fil peut être amené à attendre indéfiniment.

Par exemple : un producteur très lent, deux consommateurs très rapides : chaque fois que le producteur écrit dans le buffer, le fil consommateur n°1 récupère la donnée et reprend son attente, le fil n°2 n'accède jamais aux données.

1.2.5 Remarque

Nous allons voir plusieurs outils de synchronisation permettant l'établissement de sections critiques. Ces outils peuvent être de plusieurs natures :

- Algorithmique : nous verrons deux algorithmes permettant de travailler avec deux fils d'exécution (algorithme de PETERSEN) ou plus (algorithme de la boulangerie de LAMPORT)

  Ces algorithmes nécessitent une attente active des fils d'exécution : ils bouclent en ne faisant rien en attendant la réalisation d'une condition.

- Des primitives de programmation directement fournies par le système (mutex et sémaphore) qui permettent l'interruption d'un fil en attendant la réalisation d'une condition plutôt qu'une attente active.