Chapitre 13 : Concurrence et synchronisation

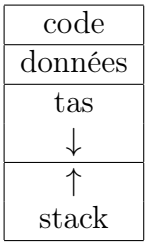
Table des matières

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

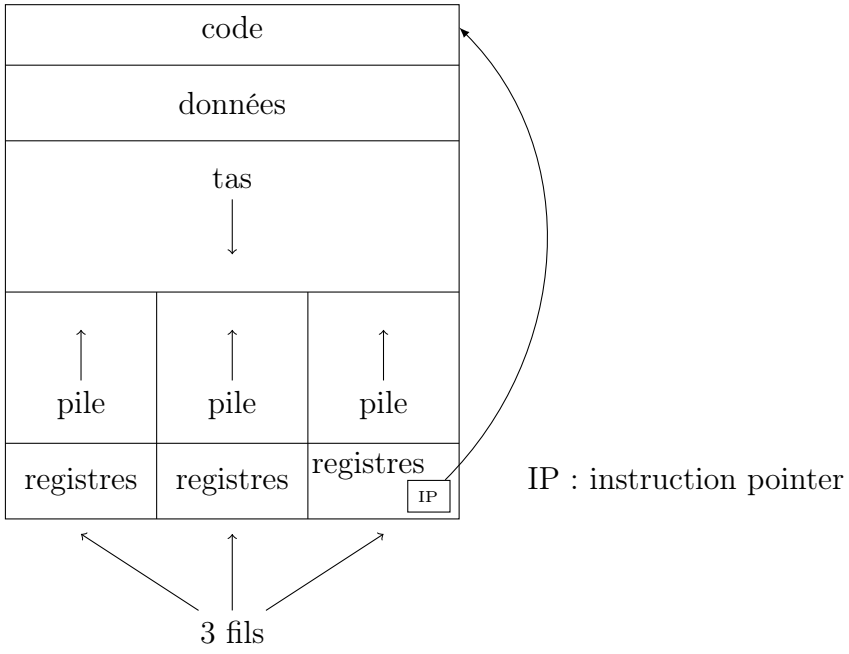
1.2.1 Fil d'exécution (*thread*)

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.

1.3 Syntaxe de la manipulation des fils d'exécution

1.3.1 En OCaml

On utilise le module `Thread`, qui propose les objets suivants :

• le type `Thread.t` qui représente les fils d'exécution ;

• la fonction `Thread.create : ('a -> 'b) -> 'a -> Thread.t` qui, étant donné une fonction `f : 'a -> 'b` et un argument `x : 'a`, crée et renvoie un fil d'exécution pour le processus courant, chargé d'évaluer `f x`. Le résultat de la fonction est ignoré, et en pratique, on utilise des fonctions de type `'a -> unit` qui font des effets de bord ;

• la fonction `Thread.join : Thread.t -> unit` qui prend un fil `p` en argument et qui interrompt le fil courant jusqu'à terminaison de `p`.

Pour compiler un programme utilisant ce module, on exécute une ligne de la forme

```
ocamlc -I +threads unix.cma threads.cma fichier.ml -o programme
```

1.3.2 En C

On inclut l'en-tête `pthread.h`.

Cela donne accès à :

• un type `pthread_t` qui représente les fils d'exécution ;

• une fonction de prototype

```c
int pthread_create(pthread_t* p, const pthread_attr_t* attr, void*
    (*f)(void*), void* args)
```

où

– `p` est un pointeur permettant de stocker le fil créé ;

– `attr` est H.P et sera toujours `NULL` ;

– `f` et `args` représentent la fonction que le nouveau fil doit exécuter et ses arguments.

Pourquoi `void*` ?

Pas de fonction d'ordre supérieur et pas de polymorphisme mais il est possible d'effectuer du transtypage vers et depuis `void*`.

On pourra se contenter de définir une fonction de type `void* f(void*)` et de la passer directement en argument. Si `f` n'a aucun argument, `args` sera `NULL`.

L'entier renvoyé est un code d'erreur qui vaut 0 si tout s'est bien passé. On adoptera un style défensif et on vérifiera la valeur de cet entier.

• La fonction de prototype

```
1 int pthread_join(pthread_t p, void** res)
```

où `p` est le fil d'exécution dont le fil courant doit attendre la terminaison, et `res` est H.P et sera toujours `NULL` (permet de récupérer le résultat de la fonction `f`).

L'entier en retour a la même signification que dans la fonction `pthread_create`.

La compilation d'un programme utilisant cette bibliothèque est de la forme :

```
gcc -lpthread file.c - o prog
```

2 Outils algorithmiques de synchronisation

2.1 Algorithme de Petersen

2.1.1 Introduction

Conçut en 1981, cet algorithme permet l'établissement d'une section critique garantissant l'exclusion mutuelle, l'absence de famine et l'absence d'interblocage.

Cependant, il nécessite une attente active de la part des fils d'exécution et est limité à deux fils concurrents, ce qui en fait un algorithme peu utilisé en pratique.

Avant de l'étudier, nous allons voir des versions plus simples, mais qui échouent.

2.1.2 Première tentative

On utilise un booléen associé à chaque fil d'exécution indiquant s'il est en section critique. Un fil souhaitant entrer en section critique attend que l'autre n'y soit plus puis met son booléen à jour. En sortie de section critique, un fil doit mettre son booléen à jour.

cf `petersen_0.c`.

Problème : si un fil est interrompu entre la fin de son attente active et la mise à jour de son booléen, il n'y a plus forcément exclusion mutuelle.

2.1.3 Deuxième tentative

Le problème de la première version vient du fait que la mise à jour du booléen est trop tardive. On pourrait plutôt signaler la volonté d'entrer en section critique : le booléen est mis à jour avant l'attente et un fil souhaitant entrer en section critique attend que l'autre fil ne veuille plus y être.

Problème : si un fil est interrompu entre la mise à jour de son booléen et le début de son attente, on peut atteindre une situation de blocage où les deux booléens valent `true`.

cf `petersen_1.c`

2.1.4 Troisième tentative

On utilise plutôt une variable indiquant quel fil d'exécution peut entrer en section critique. Lorsqu'un fil quitte la section critique, il indique que c'est au tour de l'autre fil.

cf `petersen_2.c`.

Problème : si un fil termine son exécution avant l'autre, lorsque le second fil indique que c'est au tour du fil qui s'est arrêté, son tour ne reviendra jamais, il est donc en situation de famine.

2.1.5 Algorithme de Petersen

On combine les deux dernières tentatives : un fil peut entrer en section critique si c'est à son tour ou si l'autre fil ne souhaite pas entrer en section critique.

cf `petersen_final.c`

Pseudo code :

```
turn ← 0
want ← [false, false]
Thread i :
    want[i] ← true
    turn ← 1 − i
    while want[1 − i] && turn = 1 − i
        wait
    //Critical section
    wait[i] ← false
```

• Proposition : l'algorithme de Petersen garantit l'exclusion mutuelle.

□

Par l'absurde, supposons que les fils 0 et 1 sont en section critique.

Sans perte de généralité, on suppose que le fil 0 est entré en section critique en premier.

Au moment de l'entrée en section critique du fil 1, on sait que `want[0]` vaut toujours `true` donc que `turn` vaut 1.

Ainsi, le fil 1 a du être interrompu entre l'affectation `turn` ← 0 et la boucle d'attente pour permettre l'entrée en section critique du fil 0 (s'il avait été interrompu avant, l'affectation `turn` ← 0 empêcherait son entrée en section critique).

Lorsque le fil 0 effectue l'affectation `turn` ← 1, il ne peut plus entrer en section critique car `want[1]` vaut `true` : absurde ■

• Proposition : l'algorithme de PETERSEN garantit le progrès de l'exécution et le temps d'attente borné pour les fils d'exécution.

□

Supposons que le fil 0 veuille entrer en section critique.

Si le fil 0 est en attente, c'est que `want[1]` vaut `true`, et `turn` vaut 1.

Dans ce contexte, le fil 1 peut être dans plusieurs états :

– le fil 1 se situe juste après l'affectation `want[1]` ← `true`. Dans ce cas, on a l'affectation `turn` ← 0, le fil 1 se met en attente (on a aussi `want[0]` qui vaut `true`) et le fil 0 peut entrer en section critique ;

– le fil 1 se situe à la boucle d'attente entre le moment où le fil 0 effectue l'affectation `want[0]` ← `true` et l'affectation `turn` ← 1.

Dans ce cas, lorsqu'il reprend son exécution, le fil 1 entre en section critique. Lorsqu'il sort de section critique, on a `want[1] = false` et il y a deux possibilités :

(1) le fil 0 reprend son exécution et entre en section critique ;

(2) le fil 1 poursuit son exécution et tente à nouveau d'entrer en section critique.

`want[1]` redevient `true`, mais le fil 1 finira par exécuter l'affectation `turn` ← 0 et le fil 1 se met en attente et le fil 0 entre en section critique

– le fil 1 se situe à la boucle d'attente avant l'affectation `want[0]` ← `true`.

Le fil 1 est donc en section critique, et comme avant, ne fera qu'un passage en section critique avant de céder la place au fil 0.

Le même raisonnement s'applique au fil 1 par symétrie.

On en déduit que l'exécution progresse toujours et qu'un fil d'exécution souhaitant entrer en section critique attendra au plus un passage de l'autre fil en section critique. ■

• Remarque : la proposition précédente assure l'absence de famine, donc d'interblocage.

2.2 Algorithme de la boulangerie de LAMPORT

2.2.1 Introduction

Conçut en en 1974, cet algorithme permet également la mise en place d'une section critique avec les mêmes propriétés.

Il est aussi basé sur l'attente active, mais permet l'utilisation de plus de 2 fils.

Il existe des solutions plus efficaces que donc algorithme n'a qu'un intérêt théorique.