Chapitre 13 : Concurrence et synchronisation

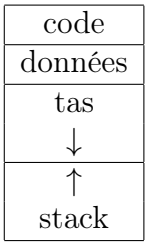
Table des matières

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 d'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 des

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

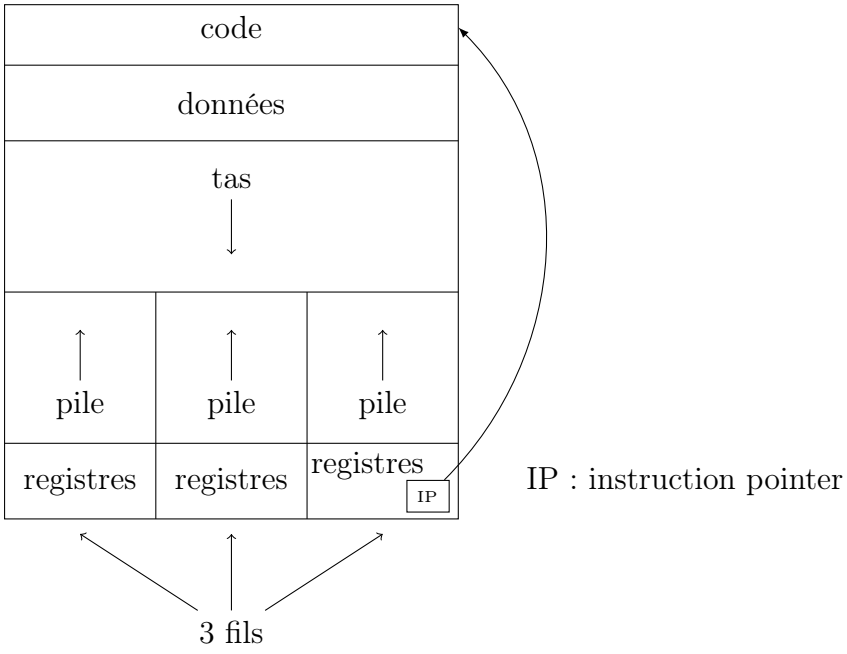
1.2.1 Fil d'exécution (*thread*)

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 fil 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 3) est une situation de compétition que l'on peut 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

```
1 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
```

ou

```
gcc -pthread 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.

cf `petersen_1.c`

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`.

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 :

Algorithm 1: Petersen

```
1 turn ← 0;
2 want ← [false, false];
3 Thread i :
4     want[i] ← true;
5     turn ← 1 − i;
6     while want[1 − i] && turn = 1 − i do
7         Wait
      // Critical section
8     want[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` $\leftarrow 0$ et la boucle d'attente pour permettre l'entrée en section critique du fil 0 (s'il avait été interrompu avant, l'affectation `turn` $\leftarrow 0$ empêcherait son entrée en section critique).

Lorsque le fil 0 effectue l'affectation `turn` $\leftarrow 1$, 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]` $\leftarrow$ `true`. Dans ce cas, on a l'affectation `turn` $\leftarrow 0$, 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]` $\leftarrow$ `true` et l'affectation `turn` $\leftarrow 1$.

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` $\leftarrow 0$ 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]` $\leftarrow$ `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.

2.2.2 Principe de l'algorithme

L'algorithme de la boulangerie de LAMPORT exploite le fonctionnement d'une file d'attente à ticket : un fil d'exécution souhaitant entrer en section critique doit prendre un ticket et attendre son tour.

Différence majeure : ce sont les fils d'exécutions qui s'attribuent eux-mêmes leur ticket. Il doivent donc déterminer quel est le numéro de ticket maximal afin de s'attribuer un numéro qui lui est supérieur.

Dans un cadre concurrent, les fils peuvent être amenés à s'attribuer le même numéro. Un fil d'exécution souhaitant entrer en section critique doit d'abord s'attribuer un numéro puis attendre d'avoir le numéro minimal, et en cas d'égalité, il faut pouvoir départager les fils. On utilise pour cela les identifiants des fils d'exécution, que l'on suppose totalement ordonnés : en cas d'égalité, le fil d'identifiant minimal a la priorité.

2.2.3 Première tentative

On utilise un tableau `ticket` qui associe à chaque fil un numéro de ticket (on suppose ici que les identifiants correspondent aux indices du tableau). L'absence de ticket sera dénoté par la valeur 0.

• Phase d'attribution du ticket pour le fil i :

$m \leftarrow \max(\texttt{ticket})$
$\texttt{ticket}[i] \leftarrow m + 1$

• Phase d'attente active pour le fil i :

On parcourt séquentiellement le tableau `ticket` en attendant pour chaque fil prioritaire devant i qu'il ait fini.

Algorithm 2: Phase d'attente active pour le fil i

```
1 for j = 0 to n − 1 do
2     while ticket[j] != 0 et (ticket[j] < ticket[i] ou (ticket[j] =
       ticket[i] et j < i)) do
3         Attendre;
  // Sortie de section critique : on détruit le ticket
4 ticket[i] ← 0 ;
```

Problème : l'exclusion mutuelle n'est pas garantie.

On suppose qu'il y a deux fils $i < j$ qui veulent entrer en section critique. *Via* des interruptions, ces deux fils peuvent calculer la même valeur pour le maximum, donc s'attribuer le même numéro. On suppose que le fil i est interrompu avant la mise à jour de son numéro et que le fil j poursuit son exécution.

Dans la phase d'attente active, le fil j constatera que `ticket[`i`] = 0`, donc poursuivra l'exécution et entrera en section critique.

Si le fil i reprend son exécution, dans la phase d'attente active il constate que le fil j a le même numéro mais n'est pas prioritaire car $i < j$, donc il poursuivra l'exécution et arrivera en section critique.

2.2.4 Algorithme de boulangerie

Le problème de la tentative précédente est qu'un fil peut être interrompu pendant la phase d'attribution, ce qui permet à des fils non prioritaires de passer devant. L'idée de LAMPORT est d'ajouter une phase d'attente supplémentaire : un fil ne peut comparer un ticket avec celui d'un autre fil que si ce dernier n'est pas en cours d'attribution.

On utilise un tableau de booléens indiquant pour chaque fil s'il est dans la phase d'attribution.

- Initialisation :

> `ticket` $\leftarrow$ tableau de taille n contenant des 0
> `attribution` $\leftarrow$ tableau de taille n contenant `false`

- Phase d'attribution pour le fil i :

> `attribution[`i`]` $\leftarrow$ `true`
> $m \leftarrow \max($`ticket`$)$
> `ticket[`i`]` $\leftarrow m + 1$
> `attribution[`i`]` $\leftarrow$ `false`

- Phase d'attente active pour le fil i :

> Pour i de 0 à $n-1$:
> Tant que `attribution[`j`]` est vrai :
> Attendre
> Tant que `ticket[`j`] != 0` et (`ticket[`j`]`, j) $<$ (`ticket[`i`]`, i)
> Attendre

Sortie de section critique pour le fil i :

> `ticket[`i`]` $\leftarrow 0$

2.2.5 Correction de l'algorithme de la boulangerie de LAMPORT

On dira qu'un fil d'exécution est dans la boulangerie entre le moment où il exécute la dernière instruction de la phase d'attribution et la fin de l'exécution de la sortie de section critique.

• Lemme 1

> *Lemme 1* : Si les fils i et j sont dans la boulangerie et si le fil i y est entré avant que le fil j exécute l'affectation `attribution[`j`]` $\leftarrow$ `true`, alors `ticket[`i`]` < `ticket[`j`]`.

□

Lorsque le fil j calcule max(`ticket`), `ticket[`i`]` contient déjà sa valeur courante (puisque i est déjà dans la boulangerie).

Donc max(`ticket`) $\geqslant$ `ticket[`i`]`

et `ticket[`j`]` $\geqslant 1 +$ `ticket[`i`]` ■

• Lemme 2

> *Lemme 2* : Si le fil i est en section critique et le fil j est dans le boulangerie avec $i \neq j$, alors
>
> $$(\texttt{ticket}[i], i) < (\texttt{ticket}[j], j)$$

□

On note t_{read} l'instant où le fil i lit pour la dernière fois `attribution[`j`]` dans la phase d'attente, et t_{wait} l'instant auquel il exécute son dernier tour dans la seconde boucle d'attente concernant le fil j.

On sait que $t_{\text{read}} < t_{\text{wait}}$.

On note :

t_{attr} l'instant auquel le fil j exécute l'affectation `attribution[`j`]` $\leftarrow$ `true` ;

t_{write} l'instant auquel le fil j a fini d'écrire la valeur de `ticket[`j`]` ;

t_{enter} l'instant auquel le fil j entre dans le boulangerie.

On sait que $t_{\text{attr}} < t_{\text{write}} < t_{\text{enter}}$.

Par définition de t_{read}, on sait que `attribution[`j`]` vaut `false` à l'instant t_{read}.

Il y a deux possibilités :

– Soit le fil j n'est pas encore dans la phase d'attribution, *i.e* $t_{\text{read}} < t_{\text{attr}}$.

Dans ce cas, comme i et j sont dans la boulangerie, et comme i y est entré avant que j commence la phase d'attribution, le lemme 1 assure que

$$\texttt{ticket}[i] < \texttt{ticket}[j],$$

donc que

$$(\texttt{ticket}[i], i) < (\texttt{ticket}[j], j)$$

– Soit le fil j a fini sa phase d'attribution à l'instant t_{read}.

On a donc $t_{\text{write}} < t_{\text{enter}} < t_{\text{read}} < t_{\text{wait}}$

Donc à l'instant t_{wait}, `ticket[`j`]` contient sa valeur courante et comme le fil i a pu poursuivre son exécution, soit `ticket[`j`]` $= 0$, soit $(\texttt{ticket}[i], i) \leqslant (\texttt{ticket}[j], j)$.

Or `ticket[`j`]` $\geqslant 1$ puisque l'attribution du ticket est déjà faite.

De plus, $(\texttt{ticket}[i], i) \neq (\texttt{ticket}[j], j)$, car $i \neq j$.

Donc

$$(\texttt{ticket}[i], i) < (\texttt{ticket}[j], j)$$

■

• Proposition

> *Proposition* : L'algorithme de la boulangerie garantit l'exclusion mutuelle.

□

Par l'absurde, on suppose que deux fils $i \neq j$ sont en section critique.

En particulier, il sont dans la boulangerie.

Par le lemme 2, on a donc :

$$\begin{cases} (\texttt{ticket}[i], i) < (\texttt{ticket}[j], j) \\ (\texttt{ticket}[j], j) < (\texttt{ticket}[i], i) \end{cases}$$

Absurde. ■

• Proposition

> *Proposition* : L'algorithme de la boulangerie garantit le progrès de l'exécution.

□

Par l'absurde, on suppose qu'au moins un fil souhaite entrer en section critique, que la section critique est libre, et qu'on peut retarder indéfiniment l'entrée d'un fil en section critique.

Comme la phase d'attribution n'est pas bloquante, à un moment tous les fils seront soit dans la phase d'attente, soit ne souhaiterons pas entrer en section critique.

Il y a au moins un fil en phase d'attente, donc on peut considérer le fil i en attente qui minimise $(\texttt{ticket}[i], i)$ (car l'ordre lexicographique est bien fondé).

Pour chaque fil j que le fil i considère dans la phase d'attente, on sait que `attribution[`j`]` vaut `false` (j est soit en phase d'attente, soit ne souhaite pas entrer en section critique), et si $\texttt{ticket}[j] \neq 0$, alors j est en phase d'attente.

Donc par définition de i, $(\texttt{ticket}[i], i) \leqslant (\texttt{ticket}[j], j)$, donc i ne peut pas être bloqué en phase d'attente : absurde. ■

• Proposition :

> *Proposition* : l'algorithme de la boulangerie garantit un temps d'attente borné.

□

Soit i un fil souhaitant entrer en section critique. On considère ce fil après la phase d'attribution, *i.e* i est dans la boulangerie.

Soit j un autre fil.

Si j commence sa phase d'exécution après l'entrée de i dans la boulangerie, le lemme 1 assure $\texttt{ticket}[i] < \texttt{ticket}[j]$, donc le fil i passera avant j en section critique.

Sinon deux possibilités : i passe avant j en section critique, soit j passe avant i.

Dans ce dernier cas, si le fil j souhaite revenir en section critique, il devra commencer une nouvelle phase d'attribution, et le lemme 1 garantit que i entrera en section critique avant le second passage de j.

Donc dans le pire cas, un fil donné devra attendre un passage de chaque fil en section critique avant de pouvoir y entrer ■

3 Outils techniques de synchronisation

3.1 Mutex

3.1.1 Introduction

Un *mutex* (pour *mutual exclusion*) est une primitive de haut niveau pour l'exclusion qui se présente sous la forme d'un verrou : un fil d'exécution souhaitant entrer en section critique doit le verrouiller, puis le déverrouiller en sortie de section critique. Si le mutex est déjà verrouillé lorsqu'un fil tente de le verrouiller, il sera mis en attente.

Remarque : on ne dit rien sur l'implémentation des mutex, qui fonctionnent comme une boite noire pour le programmeur. Les algorithmes de PETERSEN et de LAMPORT sont des implémentations possibles de mutex, même s'il existe des solutions matérielles plus efficaces.

3.1.2 Mutex en OCaml

Le module `Mutex` fournit les objets suivants :

- Le type `Mutex.t` des mutex ;
- Une fonction `Mutex.create : unit -> Mutex.t` qui sert à créer un nouveau mutex ;
- Une fonction `Mutex.lock : Mutex.t -> unit` qui verrouille un mutex donné. Si le mutex est déjà verrouillé, le fil appelant est interrompu ;
- Une fonction `Mutex.unlock : Mutex.t -> unit` qui déverrouille un mutex donné. Important : le mutex doit être verrouillé, et c'est le fil appelant qui doit l'avoir verrouillé. Les fils qui ont été interrompus parce qu'ils ont essayé de verrouiller ce mutex sont réveillés pour qu'ils tentent à nouveau de le verrouiller.

Exemple : on reprend la situation de compétition pour l'incrémentation d'une variable commune. Le fil d'exécution principal crée un mutex `m`, qu'il rend accessible aux fils secondaires. Le corps de la boucle des fils secondaires devient :

```
1 Mutex.lock m;
2 n := !n + 1;
3 Mutex.unlock m;
```

cf les fichiers `no_race_mutex.ml` et `no_race_mutex_more_threads.ml`.

3.1.3 Mutex en C

L'en-tête `pthread.h` donne accès aux objets suivants :

- Le type `pthread_mutex_t` des mutex ;
- H.P (oubli du programme) : un mutex doit être initialisé. Il y a deux options :
  - Au moment de la déclaration, on écrit une ligne de la forme

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

  - Après la déclaration sans initialisation, on utilise la fonction de prototype

```
int pthread_mutex_init(pthread_mutex_t* lock, const
    pthread_mutex_attr_t* attr);
```

    Le second paramètre est H.P, et la valeur `NULL` donne un comportement équivalent à l'initialisation directe.

    L'entier renvoyé est un code d'erreur ;

- Les fonctions de prototypes

```
int pthread_mutex_lock(pthread_mutex_t* lock);
int pthread_mutex_unlock(pthread_mutex_t* lock);
```

  qui fonctionnent comme `Mutex.lock` et `Mutex.unlock` en OCaml.

  Les entiers renvoyés sont des codes d'erreur ;

- La fonction de prototype

```
int pthread_mutex_destroy(pthread_mutex_t* lock);
```

  qui sert à détruire le mutex, *i.e* à le remettre dans un état non initialisé.

  L'entier renvoyé est un code d'erreur.

Remarque : le module `Mutex` d'OCaml est implémenté en utilisant ces fonctions.

Exemple : *cf* `no_race_mutex.c` et `no_race_mutex_more_threads.c`.

3.1.4 Retour sur le problème du dîner des philosophes

On se propose d'essayer d'utiliser les mutex pour résoudre ce problème.

- Première tentative : on pourrait utiliser un mutex global concernant l'activité `manger`.

  Code :

  - Déclaration globale :

```
pthread_mutex_t diner = PTHREAD_MUTEX_INITIALIZER;
```

  - Fonction exécutée par chaque philosophe :

```c
void* philosopher(void* args) {
    int i = *((int*) args);
    printf("Philosopher %d is starting\n", i);

    while (true) {
        printf("Philosopher %d is thinking\n", i);
        pthread_mutex_lock(&diner);
        printf("Philosopher %d is eating\n", i);
        pthread_mutex_unlock(&diner);
    }
}
```

On n'oublie pas la destruction du mutex par le fil principal avant la fin de son exécution.

cf `philosopher_0.c`

Problème : on n'exploite pas toutes les possibilités du modèle concurrent car on pourrait autoriser deux philosophes à manger en même temps.

- Deuxième tentative :

  Puisqu'on veut autoriser plusieurs philosophes à manger en même temps, il faut tenir compte de la disponibilité des baguettes.

  On utilise un tableau de booléens `stick[5]`, et le philosophe i doit prendre les baguettes i et $i+1 \mod 5$ pour pouvoir manger.

  Pour éviter les situations de compétition, on protège les accès au tableau par un mutex.

  Code :

  – Déclaration globale :

```c
bool stick[5] = {true};
pthread_mutex_t array = PTHREAD_MUTEX_INITIALIZER;
```

  – Boucle du philosophe i :

```c
while (true) {
    printf("Philosopher %d is thinking\n", i);

    pthread_mutex_lock(&array);

    if (stick[i] && stick[(i + 1) % 5]) {
        stick[i] = false;
        stick[(i + 1) % 5] = false;
        pthread_mutex_unlock(&array);

        printf("Philosopher %d is eating\n", i);

        //No race situation here even if access is not protected
        stick[i] = true;
        stick[(i + 1) % 5] = true;
    }
    else
        pthread_mutex_unlock(&array);
}
```

cf `philosopher_1.c`

Problème : on reproduit de l'attente active, et il peut y avoir famine si un philosophe n'a pas de chance avec les interruptions entre ses tests.

- Troisième tentative :

  Pour éviter l'attente active, on décide d'utiliser un mutex par baguette : un philosophe souhaitant manger doit verrouiller les deux mutex correspondant à ses baguettes. L'attente active est remplacée par l'interruption jusqu'au déverrouillage du mutex.

  – Déclaration globale : on suppose déclaré et initialisé un tableau :

```
pthread_mutex_t stick[5];
```

  – Boucle du philosophe i :

```
while (true) {
    printf("Philosopher %d is thinking\n", i);

    pthread_mutex_lock(stick + i);
    pthread_mutex_lock(stick + ((i + 1) % 5));

    printf("Philosopher %d is eating", i);

    pthread_mutex_unlock(stick + i);
    pthread_mutex_unlock(stick + ((i + 1) % 5));
}
```

cf `philosopher_2.c`

Problème : il peut y avoir interblocage si chaque philosophe détient une baguette (*cf* 1.2.4, page 6).

Solution simple : on impose aux philosophes d'identifiant impair de verrouiller d'abord le mutex $(i+1) \mod 5$.

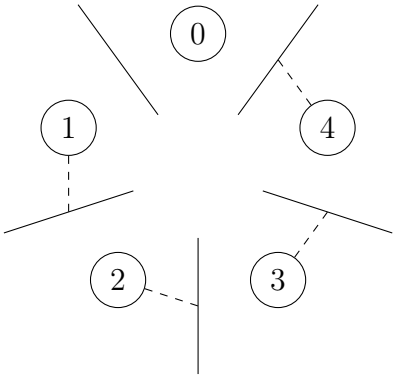
Inconvénient : cela brise la symétrie entre les philosophes. On voudrait que chaque fil d'exécution exécute le même programme.

- Autre idée :

  Pour garder la symétrie, on pourrait limiter le nombre de philosophes autorisés à tenir une baguette simultanément.

  Question : pourquoi une limite égale à quatre convient-t-elle ?

  Il reste toujours un baguette libre pour l'un de ces quatre philosophes.



On a donc besoin d'un compteur sur le nombre de philosophes tenant une baguette, protégé par une section critique pour éviter les situations de compétition et qui soit bloquant pour un philosophe qui souhaite prendre une baguette alors que les quatre autres en tiennent une. C'est impossible sans attente active avec les mutex (même problème que la deuxième tentative).

Une généralisation des mutex, appelée sémaphore, permet de gérer ce type de capteur sans attente active.

3.2 Sémaphores

3.2.1 Introduction

Conçus dans les années 1960 par Dijkstra, les sémaphores permettent de compter le nombre d'accès simultanés à une resource et de mettre en attente tout processus souhaitant accéder à la resource si le nombre maximal d'accès est atteint.

Le *sémaphore* est une primitive de synchronisation de haut niveau dont l'implémentation peut varier selon les systèmes.

On distingue le sémaphore *binaire*, qui ne permet qu'un accès à la resource et correspond donc au mutex, du sémaphore à *compteur*, qui autorise plus d'accès simultanés.

3.2.2 Définition (*sémaphore à compteur*)

Un *sémaphore à compteur* est une structure contenant :

- un compteur initialisé par l'utilisateur avec une valeur positive ;
- une file d'attente pour les fils d'exécutions interrompus.

Cette structure doit venir avec des primitives de création, d'initialisation (parfois de destruction), et de manipulation du compteur :

- Une opération nommée P : si le compteur est strictement positif, il est décrémenté et le fil poursuit son exécution. Sinon, le fil est placé dans la file d'attente ;
- Une opération nommée V : s'il existe un fil d'exécution dans la file d'attente, il est réveillé et poursuit son exécution (la ressource est devenue disponible pour ce fil). Sinon, le compteur est incrémenté.

Attention : on ne dit rien sur le fonctionnement de la file d'attente, même si un comportement proche de celui d'une file (de priorité) est désirable pour éviter les famines.

Remarque : on explique souvent les noms P et V par les termes néerlandais *proberen* (tester) et *verhogen* (incrémenter), même s'il sont souvent remplacés par d'autres termes, comme *wait* et *signal* (*cf* TD_{36}), ou *pend* et *post*.

3.2.3 Syntaxe de la manipulation des sémaphores

Même s'il existe un module `Semaphore` en OCaml, contenant deux sous-modules pour les sémaphores binaires et à compteur, le programme limite la programmation avec les sémaphores au language C.

Le fichier d'en-tête `semaphore.h` donne accès aux primitives des sémaphores à compteur (qui généralisent les sémaphores binaires).

L'en-tête donne accès aux objets suivants :

- Le type `sem_t` des sémaphores ;
- Une fonction de prototype

```
1 int sem_init(sem_t* sem, int pshared, unigned value);
```

qui initialise le sémaphore `sem` avec la valeur ($\geqslant 0$) `value`. Dans notre cas, l'entier `pshared` vaudra toujours 0 pour signaler que le sémaphore est partagé entre les fils du processus (une valeur non nulle signale que le sémaphore est partagé entre plusieurs processus, c'est donc H.P).

L'entier renvoyé est un code d'erreur ;

- Une fonction de prototype

```
1 int sem_destroy(sem_t* sum);
```

qui détruit un sémaphore dont la file d'attente est vide ;

- Une fonction implémentant l'opération P, de prototype

```
1 int sem_wait(sem_t* sem);
```

Différence avec 3.2.2 : le compteur est toujours décrémenté, et le fil appelant est placé dans un fil d'attente si la nouvelle valeur du compteur est strictement négative ;

- Une fonction implémentant l'opération V, de prototype

```
1 int sem_post(sem_t* sem);
```

Différence avec 3.2.2 : le compteur est toujours incrémenté, et il existe un fil en attente qui sera réveillé si et seulement si la nouvelle valeur du compteur est négative ou nulle.

3.2.4 Application : le dîner des philosophes

On revient à l'idée évoquée en 3.1.4 (page 16) : afin d'éviter les interblocages liés aux verrous pris simultanément sur les baguettes par tous les philosophes, on limite à l'aide

d'un sémaphore le nombre de philosophes pouvant prendre un verrou à quatre.

• Initialisation :

```
sem_t pick_up;
sem_init(&pick_up, 0, 4);

pthread_mutex_t stick[5];
for (int k = 0 ; k < 5 ; k++) {
    pthread_mutex_init(&stick[i], NULL);
}
```

• Boucle du philosophe i :

```
while (true) {
    printf("Philosopher %d is thinking\n", i);

    sem_wait(&pick_up);
    pthread_mutex_lock(stick + i);
    pthread_mutex_lock(stick + ((i + 1) %5));

    //Possible to put sem_post here

    printf("Philosopher %d is eating\n", i);

    pthread_mutex_unlock(stick + i);
    pthread_mutex_unlock(stick + ((i + 1) %5));

    sem_post(&pick_up);
}
```

Remarque : ce programme garantit l'absence d'interblocage, mais l'absence de famine dépend d'une gestion équitable de la file d'attente.

cf `philosopher_sem.c`

3.2.5 Application : producteurs-consommateurs

• Rappel : on a un buffer dans lequel les producteurs écrivent des données, et les consommateurs prennent ces données.

• Besoins :

– Il faut garantir l'exclusion mutuelle pour l'accès au buffer ;

– un consommateur souhaitant prendre une donnée dans un buffer vide doit être mis en attente ;

– un producteur souhaitant écrire dans un buffer plein doit être mis en attente.

• Outils :

– l'accès au buffer est protégé par un mutex `access` ;

– la mise en attente des consommateurs est assurée par un sémaphore `full` qui compte le nombre de cases occupées : u producteur souhaitant écrire une donnée dans le buffer incrémente ce sémaphore après écriture, et un consommateur souhaitant prendre une donnée décrémente ce sémaphore avant la lecture ;

– la mise en attente des producteurs est assurée par un sémaphore `empty` qui compte le nombre de cases vides : un producteur souhaitant écrire dans le buffer décrémente ce sémaphore avant l'écriture, et un consommateur souhaitant prendre une donnée incrémente ce compteur après la lecture.

• Code :

Boucle producteur i :

```
sem_wait(&empty);
pthread_mutex_lock(&acces);
int k = rand() % 100;
pritnf("producer %d is writing %d\n", i, k);
add(k, buffer);
pthread_mutex_unlock(&access);
sem_post(&full);
```

Boucle consommateur i :

```
sem_wait(&full);
pthread_mutex_lock(&access);
int k = take(buffer);
printf("consumer %d is reading %d\n", i, k);
pthread_mutex_unlock(&access);
sem_post(&empty);
```

Remarque : un buffer est une file que l'on peut implémenter grâce à un structure contenant une liste chaînée et un pointeur vers le dernier élément de la liste (on ajoute en fin de liste et on extrait la tête) + un compteur pour vérifier la taille du buffer, ou bien à l'aide d'un tableau et de deux indices indiquant le début et la fin des données.

```
typedef struct {
    int start;
    int end;
    int buffer[SIZE];
} le_buffer_de_Gaspard;
```

cf `prod_cons.c`

3.2.6 Application : barrière de synchronisation

Une *barrière de synchronisation* pour n fils d'exécution permet de bloquer les fils sur un point de rendez-vous tant que les n fils n'y sont pas tous arrivés. Cela permet donc de resynchroniser des fils dont l'exécution ne progresse pas de la même façon.

Idée : on utilise un sémaphore initialisé à 0 que les $n-1$ premiers arrivés à la barrière décrémentent. Le dernier fil constate qu'il est dernier et incrémente $n-1$ fois le sémaphore pour relancer l'exécution de tous les fils.

Pour que le dernier fil sache qu'il est le dernier, on utilise un compteur de fil en attente protégé par un mutex.

• Initialisation :

```
sem_t barrier;
sem_init(&barrier, 0, 0);
int waiting = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- Arrivée sur la barrière :

```
pthread_mutex_lock(&lock);

if (waiting < n - 1) {
    waiting++;
    pthread_mutex_unlock(&lock);
    sem_wait(&barrier);
}
else {
    for (int i = 0 ; i < n - 1 ; i++)
        sem_post(&barrier);

    waiting = 0;
    pthread_mutex_unlock(&lock);
}
```

cf barrier.c