1 Fils d'exécution

Définition 8.1 - *processus*

Un *processus* est un programme en cours d'exécution. Il occupe la mémoire selon l'organisation suivante

- le *segment de données*, qui constitue une zone de mémoire "fixe" pour stocker les variables globales, ainsi que les constantes.
- le *tas*, où sont stockées les variables allouées dynamiquement.
- la *pile*, une zone de mémoire de taille variable où sont stockés pour chaque *fil d'exécution* :
  - le *bloc d'activation* contenant les variables locales aux fonctions appelées par le fil.
  - un *pointeur d'instruction* vers le segment de code qui contient l'adresse de la prochaine instruction du fil dans le *segment de code*.
- le *segment de code*, où sont stockés les instructions du programme.

Remarque 8.2 - *contexte d'un processus*

Deux processus ne partagent pas leur zone de mémoire. Pour communiquer, ils doivent effectuer des *appels système* (interaction avec du matériel physique comme la mémoire ou les périphériques) coûteux. Pour cette raison on dit que *le contexte d'un processus est lourd*.

Remarque 8.3

Pour les faire communiquer, on a besoin de faire des *appels système*. Par exemple, on utilise un fichier utilisé par deux processus en lecture/écriture.

Remarque 8.4

On dit que le *contexte d'un processus* est *lourd* car sa création est son activation sont coûteux.

Définition 8.5 - *fil d'exécution (thread)*

Un *fil d'exécution* (dit *thread*) est une séquence d'*instructions atomiques* : instructions s'effectuant individuellement sans interruption. On le définit par :

- sa *tâche*, sous la forme d'une fonction.
- une pile listant les intructions atomiques à traîter.

Un processus peut contenir plusieurs fils d'exécutions dont un désigné *principal*.

Remarque 8.6 - *création d'un fil d'exécution*

Pour créer un fil d'exécution, on définit sa *tâche* sous la forme d'une fonction et on lui associe une pile listant les instructions à traîter.

Remarque 8.7

Un processus peut contenir plusieurs fils d'exécutions :

- un désigné *principal*
- tous ceux créés en plus : on a autant de piles dans la mémoire que de fils d'exécution.

voir figure

Remarque 8.8 - *contexte d'un fil d'exécution*

Les fils d'exécution peuvent partager des données stockées dans le tas ou le segment de données. Ainsi, on dit que le *contexte* des fils d'exécution est *léger* car la création, l'activation et la communication est facilitée et peu coûteuse.
Pour autant, les fils d'exécution s'exécutent indépendamment les uns des autres.

Plusieurs fils d'exécution peuvent partager des données stockées dans le tas ou le segment de données.
Théoriquement, ils ont accès avec les adresses mémoires aux piles des autres fils, mais ce n'est pas utilisé en pratique.
On dit que le *contexte* des fils d'exécution est *léger* car la création, l'activation et la communication est facilitée et peu coûteuse.
Les fils d'exécutions s'exécutent indépendamment les uns les autes.

Définition 8.9 - *programmes séquentiel, concurrent*

Un *programme séquentiel* est un programme utilisant un seul fil d'exécution. Un *programme concurrent* en utilise plusieurs à la fois.

Un fil seul est *séquentiel*. On dit qu'un programme est *concurrent* lorsqu'il a plusieurs fils d'exécution (concurrent au sens de "se passe en même temps").
L'exécution repose sur un *entrelacement non déterministe* des fils d'exécution.
Dans un programme concurrent, initialement on a un fil d'exécution principal, puis on a des phases avec plusieurs fils d'execution en parallèle. Il faut explicitement attendre la terminaison des *fils secondaires* pour revenir à un unique fil principal avant la fin du processus. Voir Fig.2
Si on n'attend pas explicitement, des fils secondaires peuvent ne pas avoir terminé leur *tâche* au moment où le fil principal termine le processus.

Implémentation - *exemple de programme concurrent en pseudocode*

`t` pour tâche.

```
1  F(nom):
2      pour i=1 jusqu'à 10:
3          afficher nom
4          afficher i
5  MAIN(): // programme concurrent
6      t1 = création du fil 1 réalisant la tâche F sur "Fil 1"
7      t2 = création du fil 2 réalisant la tâche F sur "Fil 2"
8      attendre la fin de t1
9      attendre la fin de t2
10     afficher "Fin"
```

On a trois fils d'exécutions en parallèle :

- un pour la fonction `MAIN` (le *fil principal*)
- le fil 1 pour F sur "Fil1"
- le fil 2 pour F sur "Fil2"

On a plusieurs résultats possibles, issus de différents entrelacements.

À chaque exécution, on peut avoir une issue différente : c'est *non déterministe*.

Exemple 8.10

voir feuilles

Exemple 8.11

voir feuilles

En C, pour pouvoir créer des fils d'exécution, il est nécessaire d'inclure l'en-tête :

```
#include <pthread.h>
```

permettant l'utilisation de la bibliothèque POSIX `pthread`.
De plus, à la compilation, il faut ajouter l'option de compilation :

```
-pthread
```

On a un type décrivant les fils d'exécution :

```
pthread_t
```

Pour créer un fil d'exécution, on utilise la fonction de prototype suivant :

```
int pthread_create(
    pthread_t* thread,
    const pthread_attr_t* attr,
    void* (*start_routine)(void*), //transtypage
    void* arg
);
```

Avant d'appeler `pthread_create`, il faut créer un pointeur du type `pthread_t*` valide (la mémoire est allouée avant appel, mais avoir défini sur la pile convient, auquel cas on y rentre `&file`, car il lui faut quand même un pointeur, cf exemple ci-dessous).
On déclare un fil :

```
pthread_t t1; // alloué sur la pile ici
pthread_create(&t1, $\dots$); // et on y met le pointeur
```

Remarque 8.12 - *concernant le paramètre* ****start_routine***

On écrit préalablement une fonction `start_routine` qui correspond à la tâche du fil créé et qui doit être du prototype :

```
    void* start_routine(void* arg)
```

Bien qu'on ait un unique argument de type `void*`, grâce au transtypage et à l'utilisation de structures bien choisies, on peut pallier à cette restriction. De même pour le retour de la fonction `start_routine`.
Néammoins, en MPI, on ne récupère pas la sortie : en pratique, on écrit dans des variables partagées entre les fils (merci les pointeurs !).
Théoriquement, la fonction prend un pointeur vers une fonction, mais les deux syntaxes fonctionnent (attention au sujet dans ce cas).

```
    pthread_create(&t1, $\dots$, start_routine, $\dots$);
    pthread_create(&t1, $\dots$, &start_routine, $\dots$);
```

Remarque 8.13 - *concernant le paramètre* ***void* arg***

`arg` est l'argument que l'on donne à la fonction `start_routine` pour le fil créé.
Il vaut `NULL` si aucun argument n'est à transférer.

Remarque 8.14 - *concernant le paramètre* ***const pthread_attr_t* attr***

En MPI, il vaut `NULL`.

Remarque 8.15 - *concernant la sortie de* `pthread_create`

La valeur de retour de `pthread_create` est un entier :

- `0` si tout se passe bien.
- un code d'erreur sinon.

Mais en MPI, on ignore cette sortie.

Pour attendre la fin de l'exécution d'un fil dans un autre, (pas forcément dans le *fil principal*), on utilise la fonction :

```
1 int pthread_join(
2     pthread_t thread,
3     void** value_ptr
4 );
```

Remarque 8.16 - *paramètre* `thread`

`thread` est le fil dont on attend la terminaison. En MPI, `value_ptr` est toujours `NULL`.

Définition 8.17 - *synchronisation, et pratique*

La *synchronisation* permet de coordonner l'exécutions des fils d'exécution afin qu'ils puissent travailler ensemble de façon cohérente.
En C comme en OCaml, la fonction de synchronisation permet de bloquer l'exécution de la fonction principale (dans laquelle la fonction est appelée) pour finir l'exécution du fil d'exécution passé en argument.

Implémentation - *implémentation en C d'une démonstration de l'entrelacement non déterministe*

```c
#include <pthread.h>
#include <stdio.h>

void* F(void* arg){ // au lieu de char* str, on GÉNÉRALISE en y
// mettant directement le arg qu'on rentre dans la création du thread !
    char* chaine = (char*) arg;
    for (int i=0; i<1000; i++){
        printf("%s %d\n", chaine, i);
    }
    return NULL; // important !
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &F, "Fil1");
    pthread_create(&t2, NULL, &F, "Fil2");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Fin");
    return 0;
}
```

En OCaml, on utilise le module `Thread`. Cela ne fonctionne pas en mode interactif, il faut compiler-exécuter avec des options de compilation.

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

ocamlopt -I +threads unix.cmxa threads.cmxa ficher.ml -o fichier
```

Le type des fils d'exécution est :

```
Thread.t
```

Pour créer un fil d'exécution :

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t
```

`Thread.create f x` crée et "renvoie" un fil d'exécution que réalise la fonction `f` sur l'argument `x`.

Pour attendre la terminaison d'un fil dans un autre :

```
Thread.join : Thread.t -> unit
```

Implémentation - *implémentation en OCaml d'une démonstration de l'entrelacement non déterministe*

```
    let f chaine =
        for i=0 to 1000 do
            print_string (chaine^(string_of_int i)^"\n");
        done

    let t1 = Thread.create f "File 1 :"
    let t2 = Thread.create f "File 2 :"
    let () = Thread.join t1; Thread.join t2
    let () = print_string "Fin"
```

2 Synchronisation et concurrence

2.1 Pourquoi faut-il synchroniser ?

Pour que les fils coopèrent sur des variables partagées, il faut leur permettre de communiquer.

Implémentation - *deux fils d'exécutions communiquantes en C*

On écrit un programme qui utilisent deux fils d'exécution qui incrémentent un *entier commun*.

```
    void* f(void* arg){
        for (int i=0; i<1000; i++){
            *((int*) arg) = *((int*) arg) + 1; // déréférencement
        }
        return NULL;
    }
    int main(){
        int x = 0;
        pthread_t t1, t2;
        pthread_create(&t1, NULL, &f, (void*) &x);
        pthread_create(&t2, NULL, &f, (void*) &x); // l'entier x est une varible par

        pthread_join(t1, NULL); // demande d'attendre t1
        pthread_join(t2, NULL); // demande d'attendre t2
        printf("x = %i \n",x);
        return 0;
    }
```

On s'attend à l'affichage de `"x = 2000"`, mais en pratique c'est plus petit.

l'incrémentation n'est pas une instance atomique :

- on a une phase de récupération de la valeur
- on a une phase d'incrémentation locale de la valeur récupérée (espace mémoire temporaire dédié)
- on a une dernière phase où l'on écrit la nouvelle valeur dans l'espace mémoire dédié.

Dans l'entrelacement, le fil actif peut changer entre deux phases. Donc la valeur écrite dans la dernière phase peut être inférieure à la valeur. Voir Fig.3

2.2 Gestion des sections critiques

Définition 8.18 - *section critique (d'après le cours)*

Une *section critique* est une suite d'instructions d'un processus nécessitant d'accéder à une ressource unique et commune à plusieurs fils d'exécution. On parle de *variable partagée*.

Définition 8.19 - *section critique*

Une *section critique* est une suite d'instructions que jamais plus d'un thread ne doit exécuter simultanément. Lorsqu'un fil accède à une section critique, on dit qu'il entre en *exclusion mutuelle* sur la ressource : il empêche tous les autres d'y accéder.

Enjeu : Un seul fil doit être capable d'accéder à une section critique pour une même ressource, en lecture ou en écriture.

Pour assurer cela, on peut protéger la ressource partagée avec un *verrou* que l'on peut sceller ou lever pour limiter l'accès aux sections critiques dans les différents fils. On dit qu'un fil qui est le seul à accéder à une section critique est en exclusion mutuelle.

Définition 8.20 - *exclusion mutuelle*

Lorsqu'un fil accède à une section critique, on dit qu'il entre en *exclusion mutuelle* sur la ressource : il empêche tous les autres d'y accéder.

Définition 8.21 - *course critique*

Lorsque deux fils cherchent à accéder à une même ressource et que plusieurs entrelacements donnent une issue différente, on parle de *course critique*.

Lorsque deux fils cherchent à accéder à une même ressources et que deux entrelacements au moins donnent deux résultats différents, on parle de *course critique*.

Remarque 8.22 - *concernant l'exemple de l'incrémentation mutuelle*

Dans l'exemple de la section 2.1, on obtient différentes valeurs pour la variable commune : il s'agit d'une course critique.
On cherche à éviter les courses critiques car elles causent un comportement non déterministe.

Définition 8.23 - *propriétés usuelles d'un programme concurrent*

Pour un programme concurrent, on cherche à garantir plusieurs propriétés :

1. *sûreté (ou principe de l'exclusion mutuelle)* : On a au plus un fil d'exécution accédant une section critique pour une même ressource au même moment.
2. *vivacité (ou absence de famine)* : tout fil d'exécution demandeur d'accès à une ressource partagée aura à un moment donné accès à la ressource.
3. *absence d'inter-blocage* : lorsque plusieurs fils demandent accès à une ressource partagée simultanément, au moins un obtient l'accès.

N.B : la troisième est moins forte (difficile à réaliser) que la deuxième (2 implique 3), on la met car parfois on veut juste vérifier la troisième.

Remarque 8.24 - *atomicité d'une section critique*

L'atomicité d'une section critique est importante. La mise en place de verrou de la forme suivante permet d'induire une atomicité de la section critique :

```
1    lock()
2    // section critique
3    unlock()
```

Exemple 8.25 - *de verrouillage*

Fil 1	Fil 2
`t = x`	`u = x`
`x = t+1`	`x = u+2`

La variable `x` est une ressource partagée. Les deux morceaux sont des *sections critiques pour la ressource* `x`. Dans l'état actuel, on a une course critique car les deux entrelacements donnent des résutlats différents :

Fil 1	Fil 2
`t = x`	
`x = t+1`	
	`u = x`
	`x = u+2`

La valeur de la variable `x` est augementée de 3.

Fil 1	Fil 2
`t = x`	
	`u = x`
	`x = u+2`
`x = t+1`	

La valeur de la variable `x` est incrémentée.

Remarque 8.26 - *trace d'exécution*

On peut parler de *trace d'exécution* lorsque l'on donne un entrelacement possible.

Exemple 8.27 - *élimination de section critique*

Pour ne plus avoir de section critique, on met en place un verrou `m`.

Fil 1	Fil 2
`lock(m)`	`lock(m)`
`t = x`	`u = x`
`x = t+1`	`x = u+2`
`unlock(m)`	`unlock(m)`

Le premier fil à lire l'instruction `lock(m)` verrouille `m`, le second attend alors que `m` soit déverrouillé pour lire la prochaine instruction.

Définition 8.28 - *mutex*

On met en place des verrous à l'aide de *mutex* qui disposent de deux opérations élémentaires :

- une opération de *verrouillage* : verrouille le mutex s'il est déverrouillé, ou bien attend que le mutex soit déverrouillé puis le verrouille.
- une opération de *déverrouillage* : déverrouille le mutex.

On dit qu'un mutex est une *primitive de synchronisation*, car c'est un outil utilisant des instructions simples pour gérer la synchronisation des fils d'exécutions.

Remarque 8.29 - *usage de mutex en pratique*

Dans certains langages (dont OCaml), un mutex doit être verrouillé puis déverrouillé dans un même fil. Cela reste une bonne pratique dans les autres langages.
Lorsqu'on souhaite verrouiller dans un fil et déverrouiller dans un autre, on préfèrera l'utilisation de *sémaphores* : une généralisation des mutex.

Définition 8.30 - *sémaphores binaire, à compteur*

Il existe deux types de *sémaphores* :

- les *sémaphores binaires* : c'est comme des mutex, mais qui fonctionnent entre plusieurs threads
- les *sémaphores à compteur* : une sutructre de données constituée d'un *compteur* et d'une *file d'attente*.

On initialise un sémaphore à compteur avec une file d'attente vide et un compteur initialisé à un entier positif.
Le sémaphore à compteur exploite deux opérations élémentaires :

- *demande d'accès* : notée `P` ou bien `down` (`wait` en C, `acquire` en OCaml).
  On vérifie que le compteur est strictement positif
  - si oui, one le décrémente et on passe à l'instruction suivante.
  - sinon, on attend que le compteur soit incrémenté pour un autre fil et lui applique le cas précédent dès que ça arrive.

  pendant qu'il attend, le fil d'exécution est placé dans la file d'attente du sémphore à compteur.
- *fin d'accès* : notée `V` ou bien `vp` (`post` en C, `release` en OCaml)
  Le compteur est incrémenté et si la file d'attente n'est pas vide, un des fils de la file d'attente du sémaphore (à compteur) est "réveillé" pour décrémenter le compteur et poursuivre l'exécution du dit fil. **les fils sont donc réveillés dans un ordre non déterministe**.

on fera un pseudocode de fonctionnement, la FC est lourde là

Définition 8.31 - *initialisation d'un sémaphore à compteur*

On initialise un sémaphore à compteur avec :

- une *file d'attente* vide, qui stockera les fils concernés par le sémaphore.
- un *compteur*, entier naturel non nul désignant le nombre maximal de fils pouvant accéder à la section critique protégée par le sémaphore.

Implémentation - *demande d'accès à un sémaphore*

On note `s` le sémaphore. le fil `fil` est celui dans lequel `P` est appelée.

```
1 P(s, fil):
2     si s.compteur == 0:
3         // s : "Je ne peux pas vous donner l'accès, je garde votre nom."
4         ajouter file à s.file
5         tant que compteur == 0:
6             Attendre // fil : "Ok, j'attends."
7     s.compteur = s.compteur - 1 // un emplacement libre de moins
```

Implémentation - *annonce de fin d'accès à un sémaphore*

On note $\mathcal{S}$ le sémaphore.

```
1 V(S):
2     si S.file n'est pas vide:
3         fil = defiler(S.file) // S : "C'est àvous, fil."
4         réveiller(fil) // fil : "Entendu, c'est reparti."
5     S.compteur = S.compteur + 1 // un emplacement de libéré
```

Remarque 8.32 - *rôle du compteur du sémaphore à compteur*

Le compteur joue un rôle de *compteur de ressources*.
La demande d'accès correspond à la récupération d'une ressource pour passer à la suite : on attend s'il n'y en a pas de disponible.
La fin d'accès correspond à la situation où l'on rend disponible une ressource.

Remarque 8.33 - *comparaison entre sémaphores à compteur et mutex*

Les sémaphores sont moins restrictifs que les mutexs : on peut avoir des fils qui font uniquement de la demande d'accès et des fils qui font uniquement de la fin d'accès (production de ressources).

Remarque 8.34 - *initialisation du compteur d'un sémaphore à compteur*

On peut voir dans certaines situations la valeur du compteur initiale comme le nombre maximum souhaité de fils qui peuvent exploiter les ressources partagées simultanément.

2.3 Mutex et sémaphores à compteurs en C

L'utilisation des mutex se fait à l'aide de la bibliothèque `pthread.h`. Le type utilise pour les mutex est :

`pthread_mutex_t`

On iniatialise un mutex déverrouillé de la façon suivante (pour l'avoir verrouillé direct, on le verrouille aussitôt après) :

```
1 | pthrad_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

Si la variable protégée est une variable globale, le mutex doit être aussi une variable globale (définie sur le segment de données). On ensuite trois fonctions à connaître :

- *verrouillage* :

  ```
  1 |     int pthread_mutex_lock(pthread_mutex_t* mutex);
  ```

  L'instruction `pthread_mutex_lock(&m)` verrouille `m` et renvoie 0 si tout s'est bien passé.
- *déverrouillage* :

  ```
  1 |     int pthread_mutex_unlock(pthread_mutex_t* mutex);
  ```

  L'instruction `pthread_mutex_unlock(&m)` déverrouille `m` et renvoie 0 si tout s'est bien passé.
- *destruction* :

  ```
  1 |     int pthread_mutex_destroy(pthread_mutex_t* mutex)
  ```

  L'instruction `pthread_mutex_destroy(&m)` détruit `m` (il ne peut plus être utilisé) et renvoie 0 si tout s'est bien passé.

L'utilisation des sémaphores demande l'inclusion du fichier d'en-tête `semaphore.h` :

```
1 | #include <semaphore.h>
```

Le type utilisé pour les sémaphores : est `sem_t`. On a ensuite quatre fonctions à connaître :

1. *Initialisation* :

   ```
   1 | int sem_init(
   2 |     sem_t* sem,
   3 |     int pshared,
   4 |     unsigned int value,
   5 | );
   ```

On déclare préalablement notre sémaphore et on l'initialise :

```
1 sem_t s;
2 sem_init(&s, 0, nmb)
```

avec `nmb` la valeur positive que l'on veut pour le compteur initialement (peut être 0).
`pshared` vaut 0 pour signifier que tous les fils partagent les sémaphores. Tout autre valeur relèverait du hors programme.
La valeur 0 est renvoyée si tout se passe bien.

2. *destruction* :

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

3. *demande d'accès (P, attente)* :

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

4. *fin d'accès (V, libération)* :

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

2.4 Mutex et sémaphores en OCaml

Pour les mutex, on utilise en OCaml le module Mutex. Le type utilsé est `Mutex.t`. Trois fonctions à connaître :

1. *création* :

```
1 Mutex.create : unit -> Mutex.t
```

```
1 let m = Mutex.create ()
```

`m` est un mute initialement déverrouillé

2. *verrouillage* :

```
1 Mutex.lock : Mutex.t -> unit
```

`Mutex.lock m` verrouille le mutex `m`

3. *déverrouillage* :

```
1 Mutex.unlock : Mutex.t -> unit
```

`Mutex.unlock m` déverrouille le mutex `m`.

L'utilisation des sémaphores en OCaml, n'appraît pas dans le programme mais se fait à l'aide du module `Semaphore` et des sous-modules `Semaphore.Binary` (osef, voir `Mutex` c'est pareil pour nous), et `Semaphore.Counting` (sémaphores à compteur).
Le type pour les sémaphores à compteur est `Semaphore.Counting.t`. On a trois fonctions à connaîtres :

1. *Initialisation* :

```
1 Semaphore.Counting.make : int -> Semaphore.Counting.t
```

L'appel `Semaphore.counting.make c` crée un sémaphore à compteur de compteur initial égal à `c` un entier naturel.

2. *Demande d'accès : `P`* :

```
Semaphore.Counting.acquire : Semaphore.Counting.t -> unit
```

3. *Fin d'accès : `V`* :

```
Semaphore.Counting.release : Semaphore.Counting.t -> unit
```

2.5 Algorithme de Peterson

L'algorithme de Peterson est une façon d'implémenter un mutex partagé uniquement par **2** fils d'exécution.
Cet algorithme utilise l'*attente active* : lorsqu'un fil `A` demande l'accès à une section critique, si l'autre fil `B` est déjà sur sa section critique, le fil demandeur `A` vérifie "en boucle" s'il a le droit de continuer.
C'est très coûteux et peu efficace, mais c'est la première idée mise en place historiquement pour gérer la synchronisation.
Aujourd'hui, il est possible de mettre un fil en attente passive et le réveiller, mais c'est HP.

Remarque 8.35 - *concernant l'algorithme*

un booléen suffit ne suffit pas : les fils seraient en course critique.
On va utiliser deux booléens et un entier :

- un booléen par fil qui indique si chaque fil est demandeur. Il reste `Vrai` si le fil est dans la section critique.
- un entier qui informe sur le fil prioritaire parmi les deux mis en jeu : `0` pour le fil `A` et `1` pour le fil `B`. On dit que ce fil prioritaire `a les droits d'accès`

Implémentation - *algorithme de Peterson de fonctionnement d'un mutex*

L'algorithme de Peterson est une façon d'implémenter un mutex partagé uniquement par 2 fils d'exécution.

```
flag = [false, false]  // indique si i veut accéder
tour = 0               // indique qui a les droits d'accès (prio)
lock(i):               // i est l'indice du fil.
    flag[i] = true
    tour = 1-i         // même si i veut accéder, il est poli
    tant que flag[1-i] et tour == 1-i: // i-1 attend et a la prio
        attendre       // on fait de l'attente active

unlock(i):
    flag[i] = false
```

Remarque 8.36 - *concernant l'algorithme*

- Lorsqu'un seul fil est demandeur, il n'attend pas car `flag[1-i]` est toujours valant `false`.
- Lorsqu'un fil est demandeur pendant que l'autre fil est dans la section critique. Il actualise son `flag` (son drapeau) et donne les droits d'accès à l'autre donc il attend jusqu'à que l'autre fil appelle `unlock`.
- Lorsque les deux fils demandent simultanément, ces deux drapeaux sont actualisés à `true` et `tour` prend la valeur 0 ou 1 selon l'entrelacement. Un seul des fils, celui qui a les droits d'accès, n'attend pas.
- – Le *principe d'exclusion mutuelle* est garanti : on n'a jamais les deux fils en même temps accédant à la section critique.
  – Avec une bonne utilisation de `lock` et `unlock`, à savoir un `lock` toujours suivi d'un `unlock`. On a *absence d'interblocage* : au moins un des deux fils a les droits d'accès.
  – Avec une bonne utilisation, on a également `absence de famine`. car le fil qui attend obtiendra les droits d'accès si l'autre fil est de nouveau demandeur après sa première fin d'accès.

Remarque 8.37 - *correction (caractère thread-safe) de l'algorithme de Peterson*

1. Le *principe d'exclusion mutuelle* est garanti (on n'a jamais les deux fils en même temps accédant à la section critique). En effet, lorsque les deux fils demandent simultanément, ces deux drapeaux sont actualisés à `true` et `tour` prend la valeur 0 ou 1 selon l'entrelacement. Un seul des fils accède à la section critique.
2. L'*absence de famine* est également respectée (un fil n'attend jamais indéfiniment pour l'accès) car si un fil est demandeur pendant que l'autre fil est dans la section critique, il actualise son `flag` et donne les droits d'accès à l'autre (il attend jusqu'à ce que l'autre fil appelle `unlock`).
3. L'*absence d'interblocage* (en cas de conflit, au moins un fil accède) est aussi vérifiée : Si un fil quitte la section critique (`flag[i] = false`), l'autre fil peut progresser.

2.6 Algorithme de la boulangerie (ou du BK) de Lamport

L'algorithme de la boulangerie de Lamport est une façon d'implémenter un mutex partagé par $N \in \mathbb{N}$ fil d'exécutions. N est fixé une fois pour toutes. On utilise un principe de file d'attente, voire de file de priorité.
La demande d'accès se fait en deux étapes :

1. la *phase d'acquisition* d'un numéro par le fil demandeur. Il prend (comme au BK) le numéro incrémenté du maximum des numéros déjà attribués.
2. la *phase d'attente active* pendant laquelle un fil vérifie, "en boucle" si c'est son tour *i.e.* si c'est celui avec le plus petit numéro.
   En cas d'égalité (demande d'accès simultanée), on compare les indices (différents du numéro !) des fils (dans $[\![0, N-1]\!]$).
   Attention, on utilise un marqueur booléen pour signaler qu'un fil est en *phase d'acquisition* pour attendre avant de comparer les numéros.

Remarque 8.38

Il y a beaucoup d'attente active, ce qui est coûteux et ralentit le code. Il faut retenir l'idée. Cet algorithme n'est plus utilisé tel quel.

Implémentation - *algorithme de la boulangerie de Lamport pour un mutex partagé par N fils*

Ici, `i` est l'indice du fil : `0 <= i <=` $N-1$

```
1  acqui = [false]*N // si le fil i demande un ticket
2  ticket = [0]*N // 0 -> pas de ticket assigné
3
4  lock(i):
5      // Phase d'acquisition
6      acqui[i] = true
7      num = 0
8      pour k = 0 jusqu'à N-1:
9          num = max(num, ticket[k])
10     ticket[i] = num + 1
11     acqui[i] = false
12
13     // Phase d'attente active
14     pour k = 0 jusqu'à i-1:
15         tant que acqui[k] ou (ticket[k] != 0 et ticket[k] <= ticket[i]):
16             attendre
17         // ok si k ne demande pas de ticket et
18         // k n'a pas de ticket ou un moins prioritaire=petit
19     pour k = i+1 jusqu'à N-1:
20         tant que acqui[k] ou (ticket[k] != 0 et ticket[k] < ticket[i]):
21             attendre
22
23 unlock(i):
24     ticket[i] = 0
```

Remarque 8.39

- Pendant la phase d'attente active, seul le fil d'exécution de plus petite (`ticket[i]`, `i`) n'attend pas.
  De plus, on attend si un fil est en cours d'acquisition au cas où il a récupéré un plus petit numéro mais ne l'a pas encore écrit.
- Une fois que le fil `i` a vérifié un fil `j` par rapport à `i`, il ne peut acquérir qu'un plus grand numéro que `i` car il regardera `ticket[i]` dans la phase d'acquisition.

2.7 Exmemple de situation d'interblocage avec une mauvaise situation des mutexs

On écrit un ptit programme en OCaml :

```
let m1 = Mutex.create ()
let m2 = Mutex.create ()

let f1 () =
    Mutex.lock m1;
    Mutex.lock m2;
    print_string "Section␣critique\n";
    Mutex.unlock m2;
    Mutex.unlock m1

let f2 () =
    Mutex.lock m2;
    Mutex.lock m1;
    print_string "Autre␣section␣critique\n";
    Mutex.unlock m1;
    Mutex.unlock m2

let t1 = Thread.create f1 ()
let t2 = Thread.create f2 ()
let () = Thread.join t1; Thread.join t2
```

Voici une trace d'exécution (un entrelacement) menant à un interblocage :

```
t1 : Mutex.lock m1 (*m1 est bloqué*)
t2 : Mutex.lock m2 (*m2 bloqué*)
t1 : Mutex.lock m2 (*t1 est bloqué car il attend de pouvoir bloquer m2*)
t2 : Mutex.lock m1 (*t2 est bloqué car il attend de pouvoir bloquer m1*)
```

2.8 Petit complément HP

Pour formaliser l'étude du comportement des mutex et des sémaphores partagés par deux fils d'exécution, on peut construire un *diagramme de transition*, il s'agit d'un tableau à double entrée :

- les lignes correspondent aux instructions des fils d'exécution.
- les colonnes correspondent aux instructions de l'autre.
- dans les cases, on représente à l'aide d'entiers l'état des mutex et des sémaphores.
  - Pour un mutex :
    - 1 si c'est déverrouillé
    - 0 si c'est verrouillé
    - une valeur négative si c'est impossible, on a un blocage
  - pour un sémaphore, on utilsie la valeur du compteur comme s'il pouvait être négatif : une valeur négative indique un blocage.

Pour l'exemple (i_1, i_2) état de $(\texttt{m1}, \texttt{m2})$: *à rattraper*

3 Applications et schémas de synchronisation courants

Au concours, plusieurs types d'exercices peuvent être poséés sur ce chapitre :

1. implémentation des mutex (algorithmes de Peterson et de la boulangerie de Lamport) et sémaphores (plus guidé car aucun algo n'est exigé au programme)
2. comme dans le sujet 0 des mines : utilisation de mutex (parfois sémaphores) pour paralléliser les calculs.
3. exercices théoriques de conception de schémas de synchronisation exploitant mutex et sémaphores pour répondre à une situation abstraite décrite : souvent du pseudo-code. On la travaille spécifiquement lors de cours n°4 du mardi 21/01/2024.

3.1 Attendre la fin d'une exécution.

On considère une situaton impliquant deux fils d'exécution réalisant des tâches en deux phases. On pose s un sémaphore de compteur initialement nul et on a l'entrelacement suivant :

Fil 1	**Fil 2**
Phase 1	Phase 1'
`post(s)`	`wait(s)`
Phase 2	Phase 2'

On veut que le **Fil 2** ne commence pas la Phase 2' avant que le fil 1 n'ait fini la phase 1.

Implémentation - *attendre la fin d'une exécution*

```
1  sem_t attente; // on peut pas faire le sem_init ici !
2
3  void* f1 (void* arg){
4      printf("Début␣phase␣1\n");
5      for(int i = 0; i < 1000; i++){
6          ;
7      }
8      printf("Fin␣de␣phase␣1\n");
9      sem_post(&attente);
10     printf("Début␣de␣phase␣2\n");
11     for(int i = 0; i<1000; i++){
12         ;
13     }
14     printf("Fin␣phase␣2\n");
15     return NULL;
16 }
17
18 void* f2 (void* arg){
19     printf("Début␣phase␣1'\n");
20     for(int i = 0; i < 1000; i++){
21         ;
22     }
23     printf("Fin␣de␣phase␣1'\n");
24     sem_wait(&attente);
25     printf("début␣de␣phase␣2'\n");
26     for(int i = 0; i<1000; i++){
27         ;
28     }
29     printf("Fin␣phase␣2'\n");
30     return NULL;
31 }
32
33 int main(){
34     sem_init(&attente, 0, 0);
35     pthread_t t1,t2;
36     pthread_create(&t1, NULL, &f1, NULL);
37     pthread_create(&t2, NULL, &f2, NULL);
38     pthread_join(t1, NULL);
39     pthread_join(t2, NULL);
40     sem_destroy(&attente);
41     return 0;
42 }
```

3.2 Rendez-vous ou barrière de synchronisation

On considère une situation où plusieurs fils doivent réaliser deux phases d'une tâche :

Fil
Phase 1
`--barrière de synchronisation--`
Phase 2

On souhaite qu'aucun des fils n'entame la phase 2 avant que tous les fils aient terminé la phase 1.

Implémentation - *Rendez-vous ou barrière de synchronisation*

```c
#define N 20

int compteur = 0;
pthread_mutex_t m_compteur = PTHREAD_MUTEX_INITIALIZER;
sem_t rendez_vous;

void* f(void* arg){
    printf("Début phase 1 de %d\n",*((int*)arg));
    for(int i = 0; i < 1000; i++){
        ;
    }
    printf("Fin phase 1 de %d\n",*((int*)arg));
    pthread_mutex_lock(&m_compteur);
    compteur = compteur + 1;
    if (compteur == N){
        sem_post(&rendez_vous);
    }
    pthread_mutex_unlock(&m_compteur);
    sem_wait(&rendez_vous);
    sem_post(&rendez_vous);
    // autre méthode
    // if (compteur == N){
    //     for (int i = 0; i<N; i++){
    //         sem_post(&rendez_vous);
    //     }
    // }
    // pthread_mutex_unlock(&m_compteur);
    // sem_wait(&rendez_vous);
    printf("Début phase 2 de %d\n",*((int*)arg));
    for(int i = 0; i<1000; i++){
        ;
    }
    printf("Fin phase 2 de %d\n",*((int*)arg));
    return NULL;
}

int main(){
    sem_init(&rendez_vous,0,0);
    pthread_t tab[N];
    int num[N];
    for(int i = 0; i<N; i++){
        num[i] = i;
        pthread_create(&(tab[i]),NULL,&f,&(num[i]));
    }
    for( int i = 0; i<N; i++){
        pthread_join(tab[i], NULL);
    }
    return 0;
}
```

Commentaire des implémentations (une en commentaire, une sans commentaire).
On a un compteur du nombre de fils ayant fini la phase 1. C'est une variable partagée, on la protège par un mutex.

Pour attendre le rendez-vous, on utilise un sémaphore de comtpeur initial nul (les fils ne doivent pas attendre tant que *post* n'a pas été fait).
Le N-ème fil à incrémenter le compteur doit signaler aux autres qu'ils peuvent passer à la deuxième phase :

- Soit il réveille un fil en attente (un post qui débloque un wait) et chaque fil réveille ensuite un autre avec post. Par propagation, ils se réveillent tous.
- Soit le fil fait N fois post pour donner N autorisations de passer.

3.3 Producteur-consommateur

On considère une situation où deux types de fils d'exécutions travaillent en boucle :

- les *produteurs* : Ils produisent une ressource puis la dépose dans une case disponible s'il y en a une
- les *consommateurs* : Ils prennent une des ressources disponibles et la "consomment". Une ressource n'est consommée qu'une fois, on a une nouvelle case libre après.

Implémentation - *fonctionnement du producteur*

```
1 producteur():
2     tant que vrai:
3         r = une ressource produit
4         ranger r dans une case libre
```

Implémentation - *fonctionnement du consommateur*

```
1 consommateur():
2     tant que vrai:
3         prendre une ressource dans une case pleine et la consommer
4         libérer la case
```

code à insérer.

3.4 Problème du dîner de philosophes

Des philosophes sont réunis autour d'une table ronde, chacun devant une assiette de spaghettis. Il y a une fourchettes entre deux assiettes consécutives, qui peut être utilisée par le philosophe à droite et le philosophe à gauche uniquement. *voir Fig.5* Un philosophe a besoin de deux fourchettes, la droite et la gauche, pour manger.
Les fourchettes sont des ressources partagées, pas par tous les philosophes mais uniquement par deux philosophes

adjacents.
Un philosophe alterne entre deux "états" : manger et penser. Lorsqu'il veut manger, il essaye de prendre ses deux fourchettes.

Implémentation - *première approche*

On numérote $(\Phi_i)_{i\in[\![0,\,N-1]\!]}$ les philosophes, $(\psi_j)_{j\in[\![0,\,N-1]\!]}$ les fourchettes.
On protège chaque fourchettes par un mutex : `fourchettes[i]` est un mutex pour la fourchette `i`.

```
philosophe(i): //i est le numéro du philosophe
    while(true):
        penser()
        lock(fourchettes[i]) // ça cause un problème
        lock(fourchettes[i+1 mod N])
        manger()
        unlock(fourchettes[i+1 mod N])
        unlock(fourchettes[i])
```

Dans le cas où tous les philosophes se retrouvent au niveau du problème dans le code, tous les mutex (`fourchettes[i]`) sont verrouillés et tous les fils sont obligés d'attendre.
Tous les philosophes ont pris leur fourchette droite et attendent que la fourchette gauche soit rendue disponible.
On peut représenter la situation à l'aide d'un *graphe de requêtes* (HP) $G = (S, A)$ où :

- S est l'ensemble des ressources (protégées par des mutex ou des sémaphores)
- A est un ensemble d'arêtes (a, b) telles que s'il existe un fil d'exécution qui, ayant acquis la ressource a sans l'avoir libérée, demande l'accès à la ressource b. Le graphe ne correspond pas à un quelconque entrelacement, il ne tient compte que de l'ordre des requêtes.

Avec ce protocole, le graphe de requête est le suivant. *Voir Fig.6.* Le graphe de requête est cyclique.

Remarque 8.40 - *graphe de requête*

On peut montrer que si le graphe de requêtes est acyclique, il n'y a pas d'interblocage possible.
Le cas échéant, en effectuant un tri topologique des sommets, parmi tous les fils demandeurs d'accès simultanément, il y a au moins un fil qui demande l'accès à une ressource disponible (celle qui correspond au plus grand sommet dans l'ordre topologique : le plus "vieux parent"). *ce n'est qu'une idée de démonstration*

Implémentation - *deuxième approche*

Il suffit de modifier le comportement d'un philosophe pour rendre le graphe de requête acyclique (on bouge une flèche : *Fig.6 : correction*)

```
philosophe(i):
    while(true):
        penser()
        if(i = N-1){
            lock(fourchettes[0])
            lock(fourchettes[N-1])
            manger()
            unlock(fourchettes[N-1])
            unlock(foruchettes[0])
        }
        else{
            lock(fourchettes[i])
            lock(foruchettes[i+1])
            manger()
            unlock(fourchettes[i+1])
            unlock(fourchettes[i])
        }
```

Le graphe de requête est maintenant acyclique (*Voir fig.5*) : pas d'interblocage.