

# 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 :
  - les *blocs d'activation* pour chaque appel de fonction contenant les variables locales aux fonctions.
  - un *pointeur d'instruction* vers le segment de code qui contient l'adresse de la prochaine instruction 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 et 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 instructions atomiques à traiter.

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 traiter.

### 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 autres.

### 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'exécution 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 :

```
1 | int pthread_create(  
2 |     pthread_t* thread,  
3 |     const pthread_attr_t* attr,  
4 |     void* (*start_routine)(void*), //transtypage  
5 |     void* arg  
6 | );
```

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 :

```
1 | pthread_t t1; // alloué sur la pile ici  
2 | 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 :

```
1 | 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`.

Néanmoins, 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).

```
1 | pthread_create(&t1, $\dots$, start_routine, $\dots$);  
2 | 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

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void* F(void* arg){ // au lieu de char* str, on GÉNÉRALISE en y
5  // mettant directement le arg qu'on rentre dans la création du thread !
6      char* chaine = (char*) arg;
7      for (int i=0; i<1000; i++){
8          printf("%s□%d\n", chaine, i);
9      }
10     return NULL; // important !
11 }
12
13 int main(){
14     pthread_t t1, t2;
15     pthread_create(&t1, NULL, &F, "Fil1");
16     pthread_create(&t2, NULL, &F, "Fil2");
17     pthread_join(t1, NULL);
18     pthread_join(t2, NULL);
19     printf("Fin");
20     return 0;
21 }
```

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

```
ocamllc -I +threads unix.cma threads.cma fichier.ml -o fichier
```

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

Le type des fils d'exécution est :

`Thread.t`

Pour créer un fil d'exécution :

```
1 | 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 :

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

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

```
1   let f chaine =
2     for i=0 to 1000 do
3       print_string (chaine^(string_of_int i)~"\n");
4     done
5
6   let t1 = Thread.create f "File_1:_:"
7   let t2 = Thread.create f "File_2:_:"
8   let () = Thread.join t1; Thread.join t2
9   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 communicantes en C

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

```
1   void* f(void* arg){
2     for (int i=0; i<1000; i++){
3       *((int*) arg) = *((int*) arg) + 1; // déréférencement
4     }
5     return NULL;
6   }
7   int main(){
8     int x = 0;
9     pthread_t t1, t2;
10    pthread_create(&t1, NULL, &f, (void*) &x);
11    pthread_create(&t2, NULL, &f, (void*) &x); // l'entier x est une variable par
12
13    pthread_join(t1, NULL); // demande d'attendre t1
14    pthread_join(t2, NULL); // demande d'attendre t2
15    printf("x=%i\n",x);
16    return 0;
17  }
```

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

l'incréméntation 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*

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

**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 8.19** - *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.

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

### **Remarque 8.20** - *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 8.21** - 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 8.22** - 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 : `lock()` // section critique `unlock()`

**Exemple 8.23** - de verrouillage

Fil 1	Fil 2
<code>t = x</code>	<code>u = x</code>
<code>x = t+1</code>	<code>x = u+2</code>

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 résultats différents :

Fil 1	Fil 2
<code>t = x</code>	
<code>x = t+1</code>	
	<code>u = x</code>
	<code>x = u+2</code>

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

Fil 1	Fil 2
<code>t = x</code>	
	<code>u = x</code>
	<code>x = u+2</code>
<code>x = t+1</code>	

La valeur de la variable `x` est incrémentée.

**Remarque 8.24** - *trace d'exécution*

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

**Exemple 8.25** - *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 8.26** - *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 8.27** - *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 préférera l'utilisation de *sémaphores* : une généralisation des mutex.

**Définition 8.28** - *sémaphores binaire, à compteur*

Il existe deux types de *sémaphores* :

- les *sémaphores binaires* : ce sont des mutex
- les *sémaphores à compteur* : une structure 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, on 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 sémaphore 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 décrémente 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*

**Remarque 8.29** - *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 8.30** - *comparaison entre sémaphores à compteur et mutex*

Les sémaphores sont moins restrictifs que les mutex : 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 8.31** - *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 utilisé pour les mutex est :

`pthread_mutex_t`

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

```
1 | pthread_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 utilise 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(  
2 | pthread_mutex_t* mutex;  
3 | )
```

L'instruction `pthread_mutex_unlock(m)` déverrouille `m` et renvoie 0 si tout s'est bien passé.

- *destruction* :

```
1 | int pthread_mutex_destroy(  
2 | pthread_mutex_t* mutex;  
3 | )
```

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 utilisé est `Mutex.t`. Trois fonctions à connaître :

1. *création* :

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

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

`m` est un mutex 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`.