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 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 8.13

voir feuilles

Exemple 8.14

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(
```

```
    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 8.15 - *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 8.16 - *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 8.17 - *concernant le paramètre* ***const pthread_attr_t* attr***

En MPI, il vaut `NULL`.

Remarque 8.18 - *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 :

```
int pthread_join(
    pthread_t thread,
    void** value_ptr
);
```

Remarque 8.19 - *paramètre* ***thread***

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

Implémentation - *implémentation en C du pseudocode précédent*

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

void* F(void* arg){ // au lieu de char* str, on GÉNÉRALISE en y mettant directem
    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 :

```
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 du précédent pseudocode*

```
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 communiquantes 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 varible 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'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