Chapitre 10

Concurrence

Hugo Salou MPI*

Dernière mise à jour le 13 mai 2023

Table des matières

1 Motivation

On place au centre de la classe 40 bonbons. On en distribue un chacun. Si, par exemple, chacun choisit un bonbon et, au *top* départ, prennent celui choisi. Il est probable que plusieurs choisissent le même. Comme gérer lorsque plusieurs essaient d'accéder à la mémoire ?

Deuxièmement, sur l'ordinateur, plusieurs applications tournent en même temps. Pour le moment, on considérait qu'un seul programme était exécuté, mais, le PC ne s'arrête pas pendant l'exécution du programme.

On s'intéresse à la notion de « processus » qui représente une tâche à réaliser. On ne peut pas assigner un processus à une unité de calcul, mais on peut « allumer » et « éteindre » un processus. Le programme allumant et éteignant les processus est « l'ordonnanceur. » Il doit aussi s'occuper de la mémoire du processus (chaque processus à sa mémoire séparée).

On s'intéresse, dans ce chapitre, à des programmes qui « partent du même » : un programme peut créer un « fil d'exécution » (en anglais, *thread*). Le programme peut gérer les fils d'exécution qu'il a créé, et éventuellement les arrêter. Les fils d'exécutions partagent la mémoire du programme qui les a créé.

En C, une tâche est représenté par une fonction de type `void* tache(void* arg)`. Le type `void*` est l'équivalent du type `'a` : on peut le *cast* à un autre type (comme `char*`).

```c
void* tache(void* arg) {
  printf("%s\n", (char*) arg);
  return NULL;
}

int main() {
  pthread_t p1, p2;

  printf("main: begin\n");

  pthread_create(&p1, NULL, tache, "A");
  pthread_create(&p2, NULL, tache, "B");

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  printf("main: end\n");

  return 0;
}
```

Code 1 – Création de *thread*s en C

```c
int max = 10;
volatile int counter = 0;

void* tache(void* arg) {
  char* letter = arg;
  int i;

  printf("%s begin [addr of i: %p] \n", letter, &i);

  for(i = 0; i < max; i++) {
    counter = counter + 1;
  }

  printf("%s : done\n", letter);
  return NULL;
}

int main() {
  pthread_t p1, p2;

  printf("main: begin\n");

  pthread_create(&p1, NULL, tache, "A");
  pthread_create(&p2, NULL, tache, "B");
```

```c

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  printf("main: end\n");

  return 0;
}
```

Code 2 – Mémoire dans les *thread*s en C

Dans les *thread*s, les variables locales (comme `i`) sont séparées en mémoire. Mais, la variable `counter` est modifiée, mais elle ne correspond pas forcément à $2 \times$ `max`. En effet, si `p1` et `p2` essaient d'exécuter au même moment de réaliser l'opération `counter = counter + 1`, ils peuvent récupérer deux valeurs identiques de `counter`, ajouter 1, puis réassigner `counter`. Ils « se marchent sur les pieds. »

Parmi les opérations, on distingue certaines dénommées « atomiques » qui ne peuvent pas être séparées. L'opération `i++` n'est pas atomique, mais la lecture et l'écriture mémoire le sont.

Définition : On dit d'une variable qu'elle est *atomique* lorsque l'ordonnanceur ne l'interrompt pas.

Exemple :
L'opération `counter = counter + 1` exécutée en série peut être représentée comme ci-dessous. Avec `counter` valant 40, cette exécution donne 42.

Exécution du fil A	Exécution du fil B
(1) $\text{reg}_1 \leftarrow$ `counter`	(4) $\text{reg}_2 \leftarrow$ `counter`
(2) reg_1++	(5) reg_2++
(3) `counter` $\leftarrow \text{reg}_1$	(6) `counter` $\leftarrow \text{reg}_2$

Mais, avec l'exécution en simultanée, la valeur de `counter` sera 41.

Exécution du fil A	Exécution du fil B
(1) $\text{reg}_1 \leftarrow$ `counter`	(2) $\text{reg}_2 \leftarrow$ `counter`
(3) reg_1++	(5) reg_2++
(4) `counter` $\leftarrow \text{reg}_1$	(6) `counter` $\leftarrow \text{reg}_2$

Il y a *entrelacement* des deux fils d'exécution.

Remarque (Problèmes de la programmation concurrentielle) :
- Problème d'accès en mémoire,
- Problème du rendez-vous,[a]
- Problème du producteur-consommateur,[b]
- Problème de l'entreblocage,[c]
- Problème famine, du dîner des philosophes.[d]

a. Lorsque deux programmes terminent, ils doivent s'attendre pour donner leurs valeurs.
b. Certains programmes doivent ralentir ou accélérer.
c. *c.f.* exemple ci-après.
d. Les philosophes mangent autour d'une table, et mangent du riz avec des baguettes. Ils décident de n'acheter qu'une seule baguette par personne. Un philosophe peut, ou penser, ou manger. Mais, pour manger, ils ont besoin de deux baguettes. S'ils ne mangent pas, ils meurent.

Exemple (Problème de l'entreblocage) :

Fil A	Fil B	Fil C
RDV(C)	RDV(A)	RDV(B)
RDV(B)	RDV(C)	RDV(A)

Table 1 – Problème de l'entreblocage

Comment résoudre le problème des deux incrementations ? Il suffit de « mettre un verrou. » Le premier fil d'exécution « s'enferme » avec l'expression `count++`, le second fil d'exécution attend que l'autre sorte pour pouvoir entrer et s'enfermer à son tour.

2 *Mutex*

Le mot *mutex* vient de *mutual exception* (exclusion mutuelle).

Définition :

Le type de données abstrait verrou fournit

- un type `t` : le type des verrous,
- une fonction `lock : t → unit` qui ferme le verrou,
- une fonction `unlock : t → unit` qui ouvre le verrou,
- une fonction `create : ( ) → t` qui crée un verrou,

```
1  lock(v);
2  /* section critique */
3  unlock(v);
```

de sorte que, lorsque plusieurs fils d'exécution exécutent de manière concurrent l'algorithme ci-dessous, on ait

1. au plus un seul fil d'exécution dans la section critique (exclusion mutuelle) ;
2. un fil d'exécution en attente (ayant appelé la fonction `lock`) n'empêche pas d'autres fils d'exécutions d'accéder à la section critique ;
3. un fil d'exécution ayant fini `lock` aura accès à un moment à la section critique.

2.1 À deux fils d'exécutions

On se restreint, pour simplifier, dans le cas où l'on n'a que deux fils d'exécutions.

Algorithme 1 Tentative 1 d'implémentation du type verrou

```
1 : Soit Dedans un tableau de taille 2 initialisé à F.
2 : Procédure Lock(i)        ▷ i ∈ {0, 1} est l'identifiant du fil
3 :     o ← 1 − i            ▷ identifiant de l'autre fil d'exécution
4 :     tant que Dedans[o] faire
5 :         rien
6 :     Dedans[i] ← V
7 : Procédure Unlock(i)
8 :     Dedans[i] ← F
```

Mais, cet tentative ne résout pas le problème. En effet, l'exécution où l'ordonnanceur exécute l'algorithme jusqu'à la fin de tant que pour le fil 1, puis passe au fil 2, est un contre-exemple à la tentative 1 d'implémentation du type verrou. ▷ propriété d'exclusion mutuelle

Algorithme 2 Tentative 2 d'implémentation du type verrou

```
1 : Soit Want un tableau de taille 2 initialisé à F.
2 : Procédure Lock(i)        ▷ i ∈ {0, 1} est l'identifiant du fil
3 :     o ← 1 − i            ▷ identifiant de l'autre fil d'exécution
4 :     Want[i] ← V
5 :     tant que Want[o] faire
6 :         rien
7 : Procédure Unlock(i)
8 :     Want[i] ← F
```

Cette tentative ne résout pas non plus le programme : si l'ordonnanceur exécute le fil 1 jusqu'à avant la boucle tant que, puis passe au fil 2, les deux fils sont bloqués. ▷ propriété de non-entreblocage

Algorithme 3 Tentative 3 d'implémentation du type verrou

```
1 : turn ← 0          ▷ premier fil d'exécution
2 : Procédure Lock(i)
3 :     o ← 1 − i
4 :     tant que turn = o faire
5 :         rien
6 : Procédure Unlock(i)
7 :     o ← 1 − i
8 :     turn ← o
```

Cette tentative impose une alternance 0 puis 1 puis 0… Mais, si l'un des deux fils termine, alors l'autre est bloqué.

Dans la suite, on suppose qu'un fil ne meure pas dans la section critique, ou lors de l'appel de Lock(i), ou lors de l'appel de Unlock(i).

On donne donc la tentative finale, ci-dessous, qui réussi à combler toutes les hypothèses du type verrou.

Algorithme 4 Tentative 4 d'implémentation du type verrou – Algorithme de Peterson

```
1 : turn ← 0          ▷ premier fil d'exécution
2 : Soit Want un tableau de taille 2 initialisé à F.
3 : Procédure Lock(i)
4 :     o ← 1 − i
5 :     Want[i] ← V
6 :     turn ← o
7 :     tant que turn = o et Want[o] faire
8 :         rien
9 : Procédure Unlock(i)
10 :    Want[i] ← F
```

On vérifie que les trois propriétés du type verrou sont vérifiées.

1. **Exclusion mutuelle.** Par l'absurde, supposons que les deux fils d'exécutions sont dans la section critique. Ainsi, $\text{Want}[0] = \text{Want}[1] = V$. Le fil d'exécution 0 nous donne que $\text{Want}[1] = F$ ou $\text{turn} \neq 1$. Le fil d'exécution 1 nous donne que $\text{Want}[0] = F$ ou $\text{turn} \neq 0$. On en déduit que $\text{turn} \neq 0$ et $\text{turn} \neq 1$. Or, $\text{turn} \in \{0, 1\}$ (on peut le montrer par un rapide invariant). D'où l'absurdité.
2. **Non-interblocage.** Si les deux conditions de boucles sont vraies, alors $\text{turn} = 0$ et $\text{turn} = 1$, ce qui est absurde.
3. **Résilience à la mort de l'autre fil d'exécution.**[1] Supposons que le fil d'exécution n'exécute plus Lock(1) (mais il a exécuté Unlock(1) avant de partir). Alors, $\text{Want}[1] = F$, et ce pour toujours. Donc, le fil 0 n'est pas bloqué.

1. *i.e.* accès peu importe si l'autre est encore en vie ou non.

2.2 À N fils d'exécutions

On propose de l'algorithme de la boulangerie. On se donne un « ticket » qui donne l'ordre de passage. En voulant accéder à la boulangerie, on prend un ticket (1 plus la valeur maximale des tickets), et on attend son tour Pour attendre son tour, on attend que chacune des personnes n'ai un ticket inférieur au sien. Ceci donne l'algorithme suivant.

Algorithme 5 Tentative 1 d'implémentation du type verrou à N fils

```
1 : Ticket est un tableau de n entiers initialisés à 0.
2 : Procédure Lock(Ticket, i)
3 :     Ticket[i] ← 1 + max{Ticket[j] | j ∈ ⟦0, n − 1⟧}
4 :     pour j ∈ ⟦0, n − 1⟧ faire
5 :         tant que Ticket[i] ≠ 0 et Ticket[j] < Ticket[i] faire
6 :             rien
7 : Procédure Unlock(Ticket, i)
8 :     Ticket[i] ← 0
```

Mais, cet algorithme peuvent prendre un ticket pendant le calcul du max. On utilise une variable EnCalcul qui dit lorsque le calcul du max est en cours.

Algorithme 6 Tentative 2 d'implémentation du type verrou à N fils

```
1 : Ticket est un tableau de n entiers initialisés à 0.
2 : EnCalcul est un tableau de n booléens initialisés à F.
3 : Procédure Lock(Ticket, i)
4 :     EnCalcul[i] ← V
5 :     Ticket[i] ← 1 + max{Ticket[j] | j ∈ ⟦0, n − 1⟧}
6 :     EnCalcul[i] ← F
7 :     pour j ∈ ⟦0, n − 1⟧ faire
8 :         tant que EnCalcul[j] faire
9 :             rien
10 :        tant que Ticket[i] ≠ 0 et Ticket[j] < Ticket[i] faire
11 :            rien
12 : Procédure Unlock(Ticket, i)
13 :    Ticket[i] ← 0
```

Mais, cet algorithme laisse avoir deux personnes ayant un ticket de même valeur. On peut départager les deux personnes avec les identifiants.

Algorithme 7 Tentative 3 d'implémentation du type verrou à N fils – algorithme de la boulangerie

```
1 : Ticket est un tableau de n entiers initialisés à 0.
2 : EnCalcul est un tableau de n booléens initialisés à F.
3 : Procédure Lock(Ticket, i)
4 :     EnCalcul[i] ← V
5 :     Ticket[i] ← 1 + max{Ticket[j] | j ∈ ⟦0, n − 1⟧}
6 :     EnCalcul[i] ← F
7 :     pour j ∈ ⟦0, n − 1⟧ faire
8 :         tant que EnCalcul[j] faire
9 :             rien
10 :        tant que Ticket[j] ≠ 0 et [Ticket[j] < Ticket[i] ou (Ticket[j] = Ticket[i] et j < i)] faire
                                    ordre lexicographique
11 :            rien
12 : Procédure Unlock(Ticket, i)
13 :    Ticket[i] ← 0
```

Cet algorithme assure l'exclusion mutuelle. On peut penser que ce problème peut créer le problème de famine. Mais, non, la comparaison entre identifiants n'a lieu qu'en cas d'égalité de ticket, donc lorsque deux personnes arrivent en même temps à la boulangerie.

3 Sémaphore

Un sémaphore est utilisé pour assurer une propriété moins forte que le *mutex* : on assure qu'il n'y a pas « trop » de personnes dans la zone critique. On fixe un nombre maximal de fils qui sont dans la zone critique et on évite un « flot » de personnes ininterrompu dans la zone critique. Par exemple, en TP, on n'a qu'un nombre limité d'ordinateurs. À l'entrée de la salle, on met à disposition les ordinateurs et chacun dépose le sien lorsqu'il a fini de l'utiliser.

Définition : Le type de données abstrait sémaphore fournit

- le type `t` des sémaphores,
- une fonction d'acquisition du sémaphore `acquire` : $\texttt{t} \to ()$,
- une fonction de libération du sémaphore `release` : $\texttt{t} \to ()$,
- une fonction de création/d'initialisation du sémaphore `make` : $\mathbb{N} \to \texttt{t}$,

tels que

- lors d'une tentative d'acquisition du sémaphore : si le compteur du sémaphore est nul, alors le fil d'exécution courant est mis en attente ; sinon, le compteur est décrémenté et le fil d'exécution peut continuer ;
- lors de la libération du sémaphore : si un fil d'exécution est en attente, on le laisse continuer son exécution ; sinon, on incrémente le compteur.

EXEMPLE :
On considère deux « types » de personnes. Les *producteurs* qui peuvent écrire une donnée. Les *consommateurs* qui récupère une donnée (qui consomme une donnée). En tant que métaphore, on considère que les données sont des pommes. Le producteur ne doit pas produire trop de pommes, le consommateur doit avoir des pommes à consommer. On considère, à présent, l'algorithme ci-dessous.

Algorithme 8 Utilisation de la structure SÉMAPHORE dans une problématique producteur/consommateur

```
1 : Procédure Producteur
2 :     acquire plein
3 :     lock()
4 :     Produit une pomme.
5 :     unlock()
6 :     release vide

7 : Procédure Consommateur
8 :     acquire vide
9 :     lock()
10 :    Mange une pomme.
11 :    unlock()
12 :    release plein

13 : vide ← make 0
14 : plein ← make nb_pommes
15 : pour i ∈ ⟦1, n⟧ faire
16 :     Soit un nouveau consommateur.      ▷ i.e. un fil d'exécution qui exécute la
         procédure Consommateur
17 : pour j ∈ ⟦1, m⟧ faire
18 :     Soit un nouveau producteur.        ▷ i.e. un fil d'exécution qui exécute la procédure
         Producteur
```