Chapitre 8 : Concurrence

Les différentes formes de concurrence. Informellement, la notion de concurrence correspond au fait que plusieurs fils d'exécution s'exécutent de manière non indépendante (par exemple en partageant des données) les uns des autres. Citons quelques exemples de concurrence.

- La machine sur laquelle on exécute un programme, n'est pas exclusivement en train d'exécuter ce programme, il lui faut aussi gérer (par exemple) les déplacements de la souris, les retours graphiques, ….
- Certains calculs coûteux (phase d'apprentissage d'un LLM♣ , simulation numérique coûteuse, …) nécessitent que les calculs soient répartis sur plusieurs machines qui interagissent pour produire le résultat final.
- Le réseau internet est un ensemble de machines, opérant ensemble sur les données que sont les pages web.

1 Vocabulaire de la concurrence

Vocabulaire 1.1

*Un **programme concurrent** est un ensemble de programmes séquentiels "classiques". Ces programmes sont composés d'**instructions atomiques**, qui est une instruction dont l'exécution ne peut être scindée : une fois l'exécution de l'instruction commencée, celle-ci se poursuit sans être interrompue.*

Exécution d'un programme concurrent. L'exécution d'un programme concurrent est non déterministe. Une exécution possible d'un programme concurrent, appelée **scénario**, est obtenue en entrelaçant les différentes exécutions des programmes séquentiels qui le compose.

Remarque 1.2

Dans nos machines, la création de cet entrelacement est déléguée à l'**ordonnanceur**, qui est un programme s'exécutant sur la machine et chargé de répartir les ressources de calcul entre les différents fils d'exécution en cours d'exécution.
Le modèle d'exécution adopté ici (où tous les entrelacements sont possibles) est très permissif : il est peu probable que l'ordonnanceur donne la main au fils d'exécution P pour une seule instruction élémentaire.
Toutefois dans un objectif de généralité nous ne faisons aucune hypothèse sur les entrelacements possibles.

Remarque 1.3

Lorsque les programmes séquentiels composant un programme concurrent exécutent une boucle infinie on ne considère que des entrelacements faisant intervenir chaque fil d'exécution infiniment souvent. Autrement dit : lorsque plusieurs programmes séquentiels s'exécutent de manière concurrente, on demande à ce que l'ordonnanceur passe la main à chacun de ces programmes séquentiels après une durée finie.

♣. *large language model*

Vocabulaire 1.4

*On appelle **fil d'exécution** l'un des programmes séquentiels constituant le programme concurrent. Étant donné un fil en cours d'exécution, on appelle **pointeur d'instruction** la prochaine instruction qu'il doit exécuter.*

Fil d'exécution P	Fil d'exécution Q
1 p_1 ; 2 p_2 ;	1 q_1 ; 2 q_2 ; 3 q_3 ;

Algorithme 1 – Exemple de programme concurrent à deux fils d'exécution P et Q.

Exemple 1.5

L'exécution du programme concurrent de l'algorithme 1 ci-dessous peut conduire au scénario suivant : $q_1 \to q_2 \to p_1 \to q_3 \to p_2$. Après avoir exécuté les instructions atomiques p_1, q_1 et q_2, les pointeurs d'instructions de P et Q indiquent respectivement les instructions p_2 et q_3.

Les autres scénarios du programme concurrent de l'algorithme 1 sont :

- $p_1 \to p_2 \to q_1 \to q_2 \to q_3$;
- $p_1 \to q_1 \to p_2 \to q_2 \to q_3$;
- $p_1 \to q_1 \to q_2 \to p_2 \to q_3$;
- $p_1 \to q_1 \to q_2 \to q_3 \to p_2$;
- $q_1 \to p_1 \to p_2 \to q_2 \to q_3$;
- $q_1 \to p_1 \to q_2 \to p_2 \to q_3$;
- $q_1 \to p_1 \to q_2 \to q_3 \to p_2$;
- $q_1 \to q_2 \to p_1 \to p_2 \to q_3$;
- $q_1 \to q_2 \to q_3 \to p_1 \to p_2$.

Exercice de cours 1.6

Justifier que tous les entrelacements possibles pour l'exemple ci-dessus ont été envisagés.
Autrement dit justifier qu'il y a seulement 10 entrelacements possibles ici.

Notation 1.7

*Dans la suite, les entrelacements obtenus lors de l'exécution concurrente de deux fils d'exécution P et Q seront notés en donnant : le nom du fil d'exécution (par exemple **P**) suivi de la ligne de programme que ce fil a exécuté. Le premier scénario de l'exemple ci-dessus sera alors noté **Q1**, **Q2**, **P1**, **Q3**, **P2**.*

Diagramme d'états. L'ensemble des exécutions possibles d'un programme concurrent peut être représenté au moyen d'un **diagramme d'états**. Un tel diagramme est un graphe orienté dont les sommets sont les **états**♣ de l'exécution du programme (valeurs des pointeurs d'instructions et état de la mémoire), deux états e et e' sont alors reliés par un arc dès lors qu'une étape d'exécution du programme concurrent peut conduire de l'état e à l'état e'.

Exemple 1.8

Considérons le programme concurrent ci-dessous, dans lequel deux fils d'exécution P et Q opèrent sur n, une variable globale entière initialisée à 0, comme indiqué dans le préambule de l'algorithme.

♣. On se limite aux états accessibles

$n \leftarrow$ variable globale initialisée à 0 ;	
Fil d'exécution P	Fil d'exécution Q
1 $n \leftarrow 1$;	1 $k_2 \leftarrow$ variable locale init. à 2 ; 2 $n \leftarrow k_2$;

Algorithme 2 – Algorithme exemple

Un état de l'exécution de cet algorithme est représenté par les éléments suivants.

- Deux entiers : les pointeurs d'instructions, (• représente un fils d'exécution ayant terminé son exécution). Par exemple $(1, 2)$ représente un état où la prochaine instruction à exécuter pour P est $n \leftarrow 1$ et la prochaine instruction à exécuter pour Q est $n \leftarrow k_2$.
- Un environnement portant sur les variables du programme. Par exemple $(n \mapsto 2, k_2 \mapsto 2)$ représente un environnement dans lequel n vaut 2 et k_2 vaut 2.

Le diagramme d'états de cet algorithme est alors le suivant.

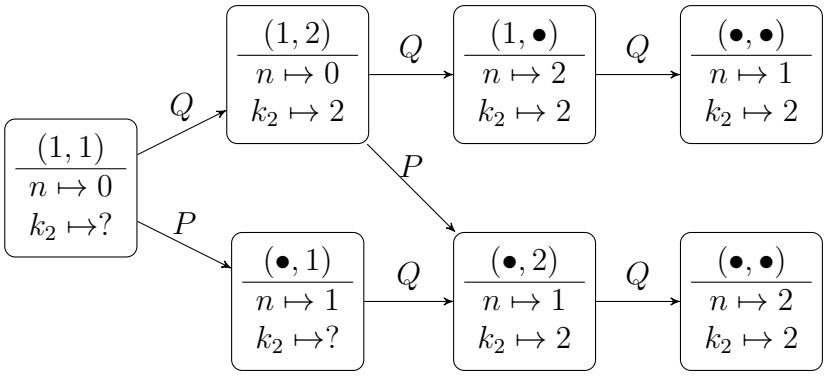

FIGURE 1 – Diagramme d'états de l'algorithme 2

2 Atomicité

Afin de rendre plus clairs les algorithmes, on se permet généralement d'utiliser des instructions élémentaires "de haut niveau"♣. Dans ce chapitre, on prendra garde à n'utiliser comme opérations élémentaires que des instructions atomiques, afin de ne pas négliger d'éventuels entrelacements. Afin d'illustrer cette problématique on considère dans cette section le cas de l'instruction d'incrémentation `c++`.

Exemple de l'incrémentation. L'instruction d'incrémentation `c++` du langage C est compilée en langage bas niveau en trois instructions. On donne ci-dessous ces trois instructions en assembleur et leur équivalent en pseudo-code.

```
mov     0x2d78(%rip),%eax # 4070 <c>
add     $0x1,%eax
mov     %eax,0x2d6f(%rip) # 4070 <c>
```

(a) Code assembleur de l'instruction `c++`

```
1 Registre ← c;
2 Registre++;
3 c ← Registre;
```

(b) Équivalent en pseudo-code

Considérons les deux algorithmes concurrents ci-dessous.

♣. On peut penser par exemple à la condition "tant que le graphe n'est pas connexe" de l'algorithme de Kruskal, à l'instruction "Soit un sommet non encore visité" dans un algorithme de parcours de graphe

c ← variable globale initialisée à 0 ;	
Fil d'exécution P	Fil d'exécution Q
1 c++ ;	1 c++ ;

Algorithme 3 – Double incrémentation de c : version haut niveau

c ← variable globale initialisée à 0 ;	
Fil d'exécution P	Fil d'exécution Q
Soit regP var. loc. ; P1 RegP ← c ; P2 RegP++ ; P3 c ← RegP ;	Soit regQ var. loc. ; Q1 RegQ ← c ; Q2 RegQ++ ; Q3 c ← RegQ ;

Algorithme 4 – Double incrémentation de c : version bas niveau

Tous les scénarios d'exécution de l'algorithme 3 conduisent à une valeur de 2 pour la variable globale c. L'algorithme 4 peut quant à lui conduire à l'entrelacement P1, Q1, P2, Q2, P3, Q3, et donc à une valeur de 1 pour la variable globale c.

Un tel comportement peut parfois être observé en machine. Aussi, dans toute la suite du chapitre nous considérerons comme atomiques uniquement des opérations de lecture et d'écritures de valeurs "simples".

3 Programmation

Dans cette section, on répertorie les différentes fonctions C et OCaml au programme permettant la manipulation des fils d'exécution[♣]. En OCaml la manipulation de fils d'exécution se fera au moyen du module `Thread`. En C la manipulation de fils d'exécution se fera au moyen de la librairie `pthread.h`, la compilation d'un programme C utilisant la librairie `pthread.h` nécessite l'option de compilation `-pthread` : `gcc -pthread main.c -o main`.
Le schéma de manipulation des fils d'exécution est le même en OCaml et en C.

- Les fils d'exécution sont des valeurs (ayant un type propre) qui peuvent être manipulées par le programme.
- Pour créer un fil d'exécution on passe en paramètres la fonction décrivant les instructions que ce fil devra exécuter.
- L'exécution d'un fil d'exécution peut être démarrée par un appel de fonction.
- Étant donné un fil d'exécution en cours d'exécution il est possible d'attendre (de manière bloquante) que l'exécution de celui-ci termine.

Le type des fils d'exécution. En OCaml, les fils d'exécution sont des objets de type `Thread.t`. En C, les fils d'exécution sont des objets de type `pthread_t`.

Création de fils d'exécution. En OCaml, la création et le lancement de l'exécution d'un fil d'exécution se font au moyen de la fonction `Thread.create : ('a -> 'b) -> 'a -> Thread.t`. Cette

♣. *thread* en anglais.

fonction prend en paramètres une fonction `f` (de type `'a -> 'b`) et un objet `x` (de type `'a`). L'appel `(Thread.create f x)` crée alors un fil exécutant l'appel `(f x)` et retourne une valeur `p` de type `Thread.t` permettant d'identifier ce fil.
En C, la création et le lancement de l'exécution d'un fil d'exécution se font au moyen de la fonction `pthread_create` prenant 4 paramètres : un pointeur `p` vers le fils d'exécution à créer, un pointeur vers une fonction `f`, un pointeur qui ne nous intéresse pas (on mettra donc `NULL`), un pointeur vers un argument de la fonction `f`. L'appel `pthread_create(p, NULL, f, p_arg)` crée alors un fil exécutant l'appel `f(p_arg)` et écrit dans `p` un thread de type `pthread_t`. La fonction `f` doit être de signature `void* f(void*)` et `p_arg` doit donc être de type `void*`.

Attente de fin d'exécution d'un fil d'exécution. En OCaml, l'attente de fin d'exécution d'un fils d'exécution `p: Thread.t` se fait au moyen de la fonction `Thread.join : Thread.t -> unit`. Si `p` a été créé au moyen d'un appel `(Thread.create f x)`, l'appel `(Thread.join p)` met en pause l'exécution du fil d'exécution courant tant que l'appel `(f x)` n'a pas terminé.
En C, l'attente de fin d'exécution d'un fil d'exécution `p` se fait au moyen de la fonction `pthread_join` prenant en paramètres : le fil à attendre, un pointeur qui ne nous intéresse pas (on mettra donc `NULL`). Si `p` a été créé au moyen d'un appel `pthread_create(p, NULL, f, p_arg)`, alors l'appel `(pthread_join(p, NULL))` met en pause l'exécution du fil d'exécution courant tant que l'appel `f(p_arg)` n'a pas terminé.

Exemple 3.1
On fournit ci-dessous deux exemples exécutant le même algorithme concurrent, un en OCaml et un en C.

Mise au carré en OCaml.

```
(* Déclaration d'un type structuré permettant le stockage des arguments
   (nb) et de la valeur de retour (res) de la fonction au_carre. *)
type args =
  {
    nb: int           ;
    mutable res : int ;
  }

let au_carre (args: args): unit =
  (* On écrit le résultat du calcul dans de la mémoire accessible par la
     fonction appelante. *)
  args.res <- args.nb * args.nb

let () =
  let arg1 = {nb = 2; res = -1} in
  let arg2 = {nb = 9; res = -1} in
  (* On "lance" le thread pb : il doit exécuter la fonction au_carre sur
     l'argument 2 et écrire le résultat dans le champs res de arg1. *)
  let pa = Thread.create au_carre arg1 in

  (* On "lance" le thread pb : il doit exécuter la fonction au_carre sur
     l'argument 9 et écrire le résultat dans le champs res de arg2. *)
  let pb = Thread.create au_carre arg2 in

  (* On attend que les deux exécutions soient terminées. *)
  Thread.join pa;
  Thread.join pb;
  assert (arg1.res = 4 && arg2.res = 81)
```

Mise au carré en C.

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

```c
#include <assert.h>

/* Déclaration d'un type structuré permettant le stockage des arguments
 * (nb) et de la valeur de retour (res) de la fonction au_carre. */
struct args_s {
  int nb;
  int* res;
};
typedef struct args_s arg_carre;

/* La fonction que l'on souhaite exécuter de manière concurrente. */
void* au_carre(void* args) {
  /* On transtype l'argument de type void* qui est un arg_carre* */
  arg_carre* a = (arg_carre*) args;
  /* On écrit le résultat du calcul dans de la mémoire accessible par la
   * fonction appelante. */
  *(a->res) = a->nb * a->nb;
  return NULL;
}

int main(){
  int resA, resB;
  arg_carre argsA = {2, &resA};
  arg_carre argsB = {9, &resB};
  /* Déclaration des deux threads */
  pthread_t pA, pB;
  /* On "lance" le thread pA : il doit exécuter la fonction au_carre sur
   * l'argument 2 et écrire la valeur résultat dans resA. */
  pthread_create(&pA, NULL, au_carre, &argsA);

  /* On "lance" le thread pB : il doit exécuter la fonction au_carre sur
   * l'argument 9 et écrire la valeur résultat dans resB. */
  pthread_create(&pB, NULL, au_carre, &argsB);

  /* On attend que les deux exécutions soient terminées. */
  pthread_join(pA, NULL);
  pthread_join(pB, NULL);
  assert((resA == 4) && (resB == 81));
  return 0;
}
```

4 Exclusion mutuelle

4.1 Définition du problème et verrou

Le problème de l'**exclusion mutuelle** est un problème classique dans le domaine de la concurrence. Afin d'illustrer ce problème, reprenons l'exemple de la double incrémentation de la section précédente. Deux fils d'exécution souhaitent incrémenter de manière concurrente un compteur partagé. La section précédente a montré qu'une telle incrémentation est "dangereuse" au sens où elle ne peut être effectuée de manière atomique et peut donc conduire à des entrelacements pour lesquelles la valeur du compteur est erronée. On souhaite donc mettre en place un mécanisme permettant d'assurer que ces entrelacements non voulus sont impossibles. On souhaite en fait assurer que si le fil d'exécution P commence l'incrémentation de c, le fil d'exécution Q ne peut pas incrémenter c tant que P n'a pas terminé. Finalement on aimerait pouvoir désigner une zone du code de P et une zone du code de Q dans lesquelles les P et Q ne peuvent se trouver de manière simultanée : on appelle cette propriété l'**exclusion mutuelle** et ces zones de code les **sections critiques**.
Cette situation est résumée dans l'algorithme 5.

Fil d'exécution P	Fil d'exécution Q

tant que … **faire**
  Section non critique ;
  Section critique ;
  Section non critique ;
	tant que … **faire**
	  Section non critique ;
	  Section critique ;
	  Section non critique ;

Algorithme 5 – Le problème de la section critique

Cadre de résolution. On considère un programme concurrent où N fils d'exécution exécutent en boucle une séquence d'instructions (chacun la leur), parmi lesquelles certaines sont identifiées comme formant une section critique. On cherche alors à mettre en place un protocole (un ensemble d'instructions) à suivre par les fils d'exécution, en deux temps (avant la section critique, après la section critique), qui assurent les trois propriétés ci-dessous.

1. Les instructions des sections critiques de deux fils d'exécution ne peuvent pas être entrelacées, on appelle cette propriété l'**exclusion mutuelle**.
2. Si un fil d'exécution souhaite accéder à la section critique ; alors il n'empêche pas l'exécution des autres fils d'exécution, on appelle cette propriété l'**absence d'interblocage**.
3. Si un fil d'exécution souhaite accéder à sa section critique, alors il pourra éventuellement y accéder, on appelle cette propriété l'**absence de famine**.

On fait de plus les suppositions suivantes.

- Les N fils d'exécution exécutent en boucle les sections non critiques et sections critiques (ainsi que les protocoles).
- L'exécution d'une section critique termine toujours et sans erreur.
- Un fil d'exécution peut être interrompu entre deux accès à la section critique.

L'algorithme 6 résume le cadre que nous nous sommes fixés dans le cas de deux fils d'exécution, les instructions soulignées désignent le protocole que l'on cherche à mettre en place.

Fil d'exécution P	Fil d'exécution Q

tant que … **faire**
  Section non critique ;
  Pré-section critique ;
  Section critique ;
  Post-section critique ;
  Section non critique ;
	tant que … **faire**
	  Section non critique ;
	  Pré-section critique ;
	  Section critique ;
	  Post-section critique ;
	  Section non critique ;

Algorithme 6 – Le problème de l'exclusion mutuelle de la section critique

Afin de rendre modulaire le protocole, on le "cache" derrière un type de donnée abstrait Verrou.

Définition 4.1

Le type de donnée abstrait Verrou, fournit une définition de type verrou et trois fonctions de manipulations de ce type :

- *create :* $() \to$ *verrou, qui correspond à la création de variables globales utiles au protocole ;*
- *lock : verrou* $\to ()$*, qui correspond à la partie du protocole précédant la section critique ;*
- *unlock : verrou* $\to ()$*, qui correspond à la partie du protocole suivant la section critique.*

La figure 7 résume alors le cadre de résolution du problème de l'exclusion mutuelle que nous fixons pour la suite de cette section, dans le cas de deux fils d'exécution.

$V \leftarrow$ create() ;	
Fil d'exécution P	Fil d'exécution Q

tant que … **faire**
  Section non critique ;
  lock$(V, 0)$;
  Section critique ;
  unlock$(V, 0)$
	tant que … **faire**
	  Section non critique ;
	  lock$(V, 1)$;
	  Section critique ;
	  unlock$(V, 1)$

Algorithme 7 – Utilisation de verrou pour deux fils d'exécution

4.2 Une solution dans le cas $N = 2$, l'algorithme de Peterson

Dans cette section, on se propose de définir une implémentation du type de donnée abstrait Verrou. La version proposée ci-dessous ne permet la gestion que de deux fils d'exécution. De plus elle nécessite les fonctions lock et unlock prennent en paramètre un entier de $\{0, 1\}$ qui permet d'identifier quel fil d'exécution appelle la fonction (on numérote 0 et 1 les deux fils d'exécution).

On présente ces algorithmes (create, lock, unlock) par raffinements successifs servant d'exemples.

4.2.1 Une première mauvaise version.

On décide ici d'utiliser comme variables globales un tableau Dedans de 2 booléens, indiquant, pour chaque case d'indice $i \in \{0, 1\}$, si le fil d'exécution numéro i est, ou non, en section critique.

```
   Procedure create() :
       Soit Dedans un tableau de deux booléens initialisés à F
       retourner Dedans

   Procedure lock(Dedans, i) :
       o ←1 − i                                              // L'autre
1      tant que Dedans[o] faire Rien          // On attend tant que l'autre est en section critique
2      Dedans[i] ←V                          // On se signale comme étant en section critique

   Procedure unlock(Dedans, i) :
3      Dedans[i] ←F                    // On se signale comme n'étant plus en section critique
```

Diagramme d'états. On trouvera en figure 3 le diagramme d'états de ce premier algorithme. On remarque qu'il est possible d'atteindre une situation dans laquelle les pointeurs d'instructions de P et Q sont simultanément sur 3 et 3, (grâce à l'entrelacement P1 ; Q1 ; P2 ; Q2 ; P4 ; Q4) ce qui correspond au cas où les deux fils d'exécution sont en section critique. Ainsi le verrou ne vérifie pas la propriété 1.

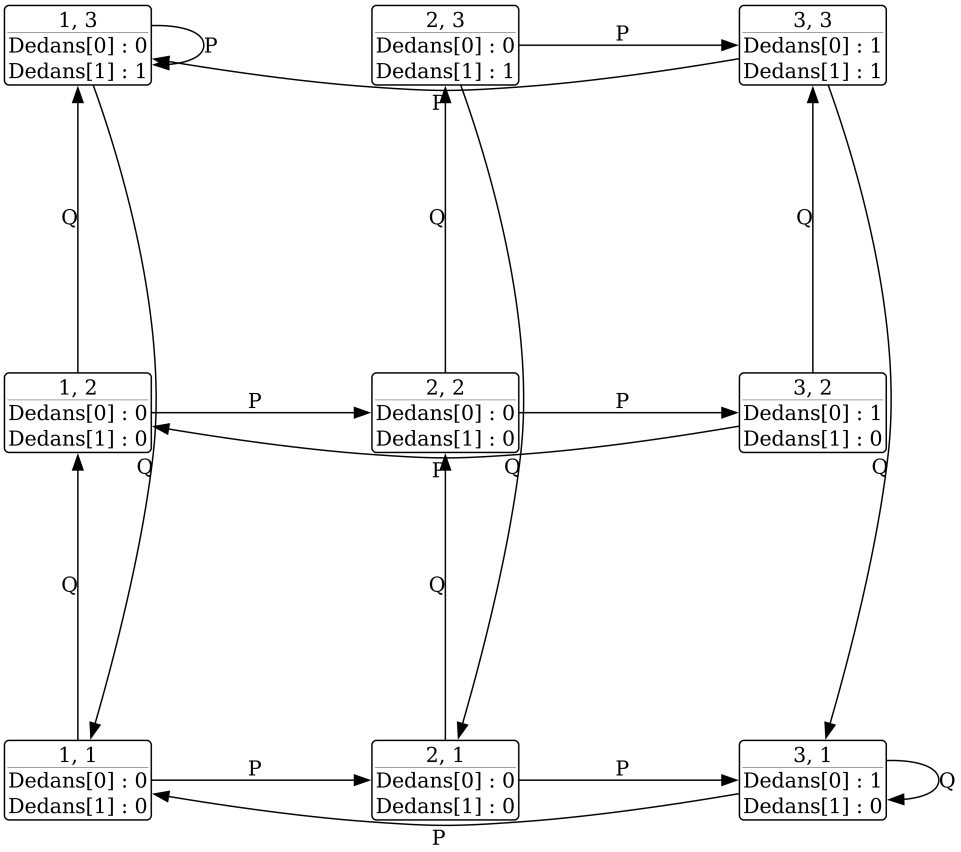

FIGURE 3 – Diagramme d'états de la première version

4.2.2 Une seconde mauvaise version.

On décide cette fois d'utiliser comme variables globales un tableau Want de 2 booléens, indiquant, pour chaque case d'indice $i \in \{0, 1\}$, si le fil d'exécution d'indice i souhaite, ou non, aller en section critique.

```
Procedure create() :
    Soit Want un tableau de deux booléens initialisés à F
    retourner Want

Procedure lock(Want, i) :
    o ←1 − i                                              // L'autre
1   Want[i] ←V                                            // On dit vouloir aller en section critique
2   tant que Want[o] faire Rien                           // On attend tant que l'autre veut aller en section critique

Procedure unlock(Want, i) :
3   Want[i] ←F                                            // On dit ne plus vouloir aller en section critique
```

Diagramme d'états. On trouvera en figure 4 le diagramme d'états de ce second algorithme. Remarquons que lorsque les pointeurs d'instruction des deux fils d'exécution sont sur les instructions 2 et 2, les fils d'exécution P et Q sont bloqués dans une boucle infinie sur cet état. Une telle boucle infinie correspond à l'entrelacement P1 ; Q1 ; P2 ; Q2 ; P3 ; Q3 ; P4 ; Q4 ; P3 ; Q3 ; ...

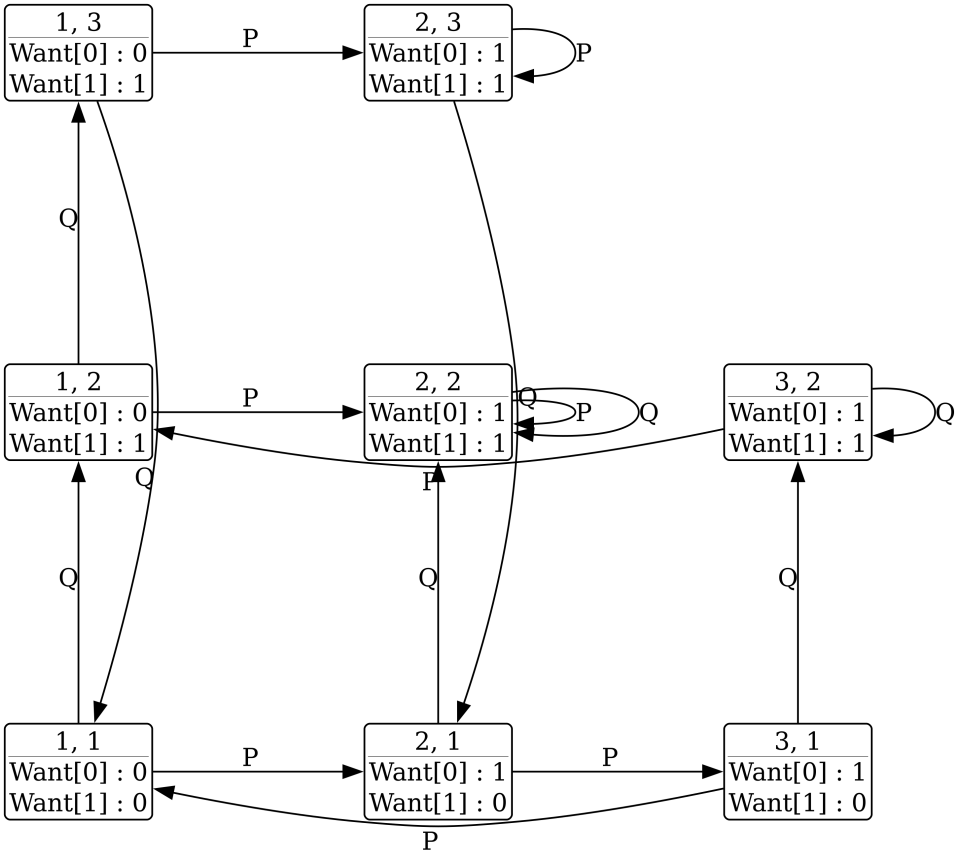

FIGURE 4 – Diagramme d'états de la seconde version

Ainsi le verrou ne vérifie pas la propriété 2.

4.2.3 Une troisième mauvaise version.

On choisit comme variables globales : une variable entière Turn valant 0 ou 1 et indiquant l'identifiant du fil d'exécution dont c'est le tour de rentrer en section critique.

Procedure create() **:**
  Soit Turn une variable entière initialisée à 0
  retourner Turn

Procedure lock(Turn, i) **:**
  $o \leftarrow 1 - i$ // L'autre
1 **tant que** Turn $= o$ **faire** Rien // On attend que ce soit notre tour

Procedure unlock(Turn, i) **:**
  $o \leftarrow 1 - i$ // L'autre
2 Turn $\leftarrow o$ // On passe le tour à l'autre

Diagramme d'états. On trouvera en figure 5 le diagramme d'états de ce troisième algorithme. On remarque que cette solution assure que les fils d'exécution rentrent alternativement en section critique : le fil d'exécution 0 a le droit de rentrer, puis le fil d'exécution 1, puis le fil d'exécution 0, ... Remarquons que lorsque les pointeurs d'instruction des deux fils d'exécution sont sur les instructions 1 et 1, si le fil d'exécution P (resp. le fil d'exécution Q) n'avance plus (comprendre le fil d'exécution

P (resp. le fil d'exécution Q) a cessé son exécution) alors le fil d'exécution Q (resp. le fil d'exécution P) ne peut plus progresser vers la section critique.

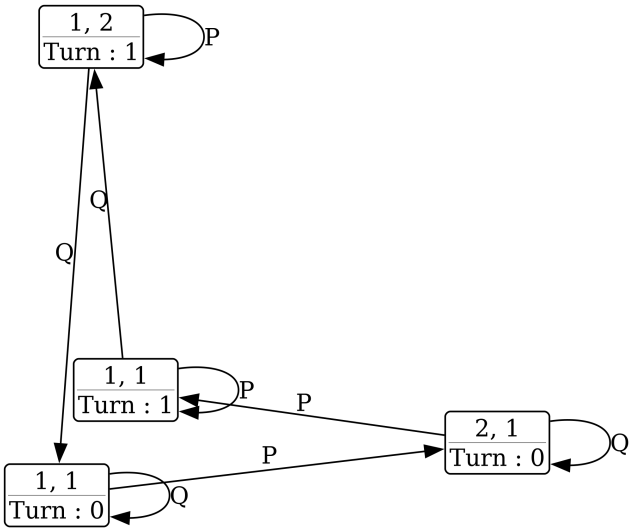

FIGURE 5 – Diagramme d'états de la troisième version

4.2.4 Version finale.

On combine les idées des deux dernières versions. On choisit comme variables globales :
- un tableau Want de 2 booléens, indiquant, pour chaque case d'indice $i \in \{0, 1\}$, si le fil d'exécution d'indice i souhaite, ou non, aller en section critique ;
- une variable entière Turn valant 0 ou 1 et indiquant l'identifiant du fil d'exécution dont c'est le tour de rentrer en section critique, cette variable sert ici seulement à décider lequel, de P ou Q, accède à la section critique en cas de demande simultanée.

```
   Procedure create() :
       Soit Turn une variable entière initialisée à 0
       Soit Want un tableau de deux booléens initialisés à F
       retourner (Turn, Want)

   Procedure lock(Turn, Want, i) :
       o ←1 − i                                          // L'autre
1      Want[i] ← V                                       // On dit vouloir aller en section critique
2      Turn ← o                                          // On cède la priorité
3      tant que Want[o] et Turn = o faire Rien           // On attend tq o veut y aller et qu'il a la priorité

   Procedure unlock(Turn, i) :
4      Want[i] ←F                                        // On dit ne plus vouloir y aller
```

Algorithme 8 – Algorithme de Peterson

Diagramme d'états. On trouvera en figure 6 le diagramme d'états de l'algorithme de Peterson.

1. On remarque qu'il n'est pas possible d'atteindre une situation dans laquelle les pointeurs d'instructions de P et Q sont simultanément sur 4 et 4, démontrant que l'exclusion mutuelle est assurée.

2. Remarquons sur le diagramme que les transitions d'un état ayant pour pointeur d'instruction $P : i$ et $Q : j$ vers un état $P : i'$ et $Q : j'$ sont de trois types.
   - Ou bien $i = 4$, $i' = 0$ et $j = j'$ (ou de même en inversant les rôles de i et j) ce qui correspond à une nouvelle itération d'accès à la section critique
   - Ou bien $i = i'$ et $j = j'$, une telle transition ne fait pas progresser l'algorithme, toutefois elle ne peut avoir lieu indéfiniment. Supposons, sans perdre en généralité, que cette transition ait lieu au moyen d'une instruction de Q. Les seules telles cas se présentent pour $i' > 0$ et $j' > 0$, ce qui assure (par hypothèse) que les deux fils d'exécution sont encore en cours d'exécution, ainsi l'entrelacement fera apparaître au moins une instruction de P dans le futur or il n'y a pas d'état présentant en même temps une boucle pour Q et pour P, ainsi éventuellement la prochaine instruction exécutée par nous fera progresser P.
   - Ou bien $i' > i$ ou $j' > j$ ce qui assure la progression de l'algorithme.

   Cette remarque nous assure l'absence d'interblocage.
3. Finalement remarquons qu'un circuit (non réduit à un état) dans le diagramme d'état faisant intervenir uniquement P (sans perdre en généralité) n'est possible que lorsque le pointeur d'instruction de Q vaut 1, signifiant que Q ne souhaite pas accéder à la section critique. Cette remarque assure l'absence de famine.

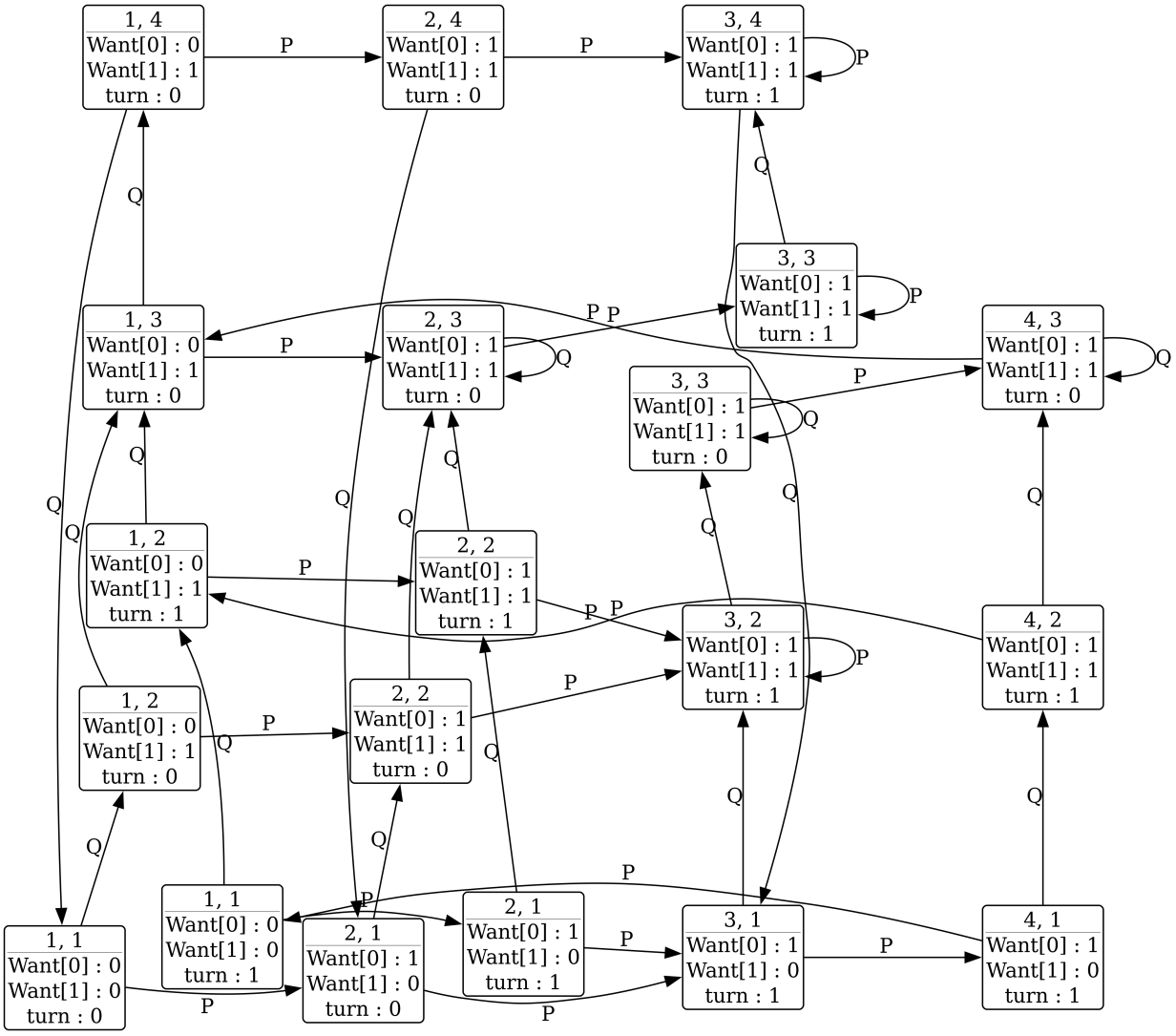

FIGURE 6 – Diagramme d'états de l'algorithme de Peterson

Proposition 4.2

L'algorithme de Peterson implémente le type abstrait verrou pour deux fils d'exécution.

Démonstration : On se place dans le cas où deux fils d'exécution P et Q utilisent le même verrou comme indiqué sur l'algorithme 7. Montrons que l'exclusion mutuelle est garantie.

Supposons par l'absurde qu'à un instant t, à la fois P et Q se trouvent en section critique. On considère pour chacun des fils les étapes du dernier appel à lock avant d'entrer en section critique, et on note alors t_i (resp. t'_i) l'instant où la ligne i de cet appel à lock a été exécutée pour la dernière fois, plus précisément :

- à l'instant t_1, P a exécuté $\mathsf{Want}[0] \leftarrow \mathsf{V}$;
- à l'instant t'_1, Q a exécuté $\mathsf{Want}[1] \leftarrow \mathsf{V}$;
- à l'instant t_2, P a exécuté $\mathsf{Turn} \leftarrow 1$;
- à l'instant t'_2, Q a exécuté $\mathsf{Turn} \leftarrow 0$;
- la dernière fois que P a évalué sa condition de boucle **tant que** se décompose en deux♣ :
  - à l'instant $t_{3.1}$, l'expression $\mathsf{Want}[1]$ est évaluée (par P) à la valeur $c_{\mathsf{W}} \in \mathbb{B}$;
  - à l'instant $t_{3.2}$, l'expression $\mathsf{Turn} = 1$ est évaluée (par P) à la valeur $c_{\mathsf{T}} \in \mathbb{B}$;

  et on sait alors que $c_{\mathsf{W}}.c_{\mathsf{T}} = \mathsf{F}(\bigstar)$;
- la dernière fois que Q a évalué sa condition de boucle **tant que** se décompose en deux :
  - à l'instant $t'_{3.1}$, l'expression $\mathsf{Want}[0]$ est évaluée (par Q) à la valeur $c'_{\mathsf{W}} \in \mathbb{B}$;
  - à l'instant $t'_{3.2}$, l'expression $\mathsf{Turn} = 0$ est évaluée (par Q) à la valeur $c'_{\mathsf{T}} \in \mathbb{B}$;

  et on sait alors que $c'_{\mathsf{W}}.c'_{\mathsf{T}} = \mathsf{F}(\bigstar')$.

Puisque chaque fil exécute ses instructions dans l'ordre, $t_1 < t_2 < t_{3.1} < t_{3.2} < t$ et $t'_1 < t'_2 < t'_{3.1} < t'_{3.2} < t$.

Afin de pouvoir nous appuyer sur la valeur de Turn pour raisonner, on travaille par disjonction de cas sur l'ordre relatif de t_2 et t'_2.

- Si $t_2 < t'_2$, alors après l'instant t'_2 on a $\mathsf{Turn} = 0$, en particulier au temps $t'_{3.1}$, donc $c'_{\mathsf{T}} = \mathsf{V}$. D'après $(\bigstar')$ on en déduit que $c'_{\mathsf{W}} = \mathsf{F}$, ce qui signifie qu'au temps $t'_{3.2}$, $\mathsf{Want}[0]$ vaut F. Pourtant $t'_{3.2} > t'_2 > t_2 > t_1$ et $t'_{3.2} < t$, donc cet instant ce situe après le moment où P a exécuté $\mathsf{Want}[0] \leftarrow \mathsf{V}$, et avant que P n'ait exécuté l'appel à unlock (seule autre instruction susceptible de modifier $\mathsf{Want}[0]$), donc $\mathsf{Want}[0]$ vaut V à l'instant $t'_{3.2}$. ABSURDE

- Si $t'_2 < t_2$, on établit de même une contradiction sur la valeur de $\mathsf{Want}[1]$.

Les deux cas étant absurdes, on en déduit qu'à aucun instant t les deux fils P et Q ne sont tous les deux en section critique. □

4.3 Une solution pour $N \geqslant 2$: l'algorithme de la boulangerie de Lamport

L'idée de cette implémentation est la suivante : on simule un distributeur de tickets, chaque fil d'exécution souhaitant rentrer en section critique prend un ticket numéroté par une valeur supérieure aux tickets des autres fils d'exécution. Le fil d'exécution ayant le ticket de plus petite valeur est autorisé à entrer en section critique. Le ticket de valeur 0 a un sens spécial : le fil d'exécution ne souhaite pas rentrer dans la section critique.

On présente l'algorithme final par raffinements successifs d'algorithmes. Dans toute la suite l'entier N est fixé et représente le nombre de fils d'exécution. Tout comme l'algorithme de Peterson on suppose ici que les fils d'exécution sont munis d'un identifiant entier unique de l'intervalle $[\![0, N-1]\!]$, et que cet entier est passé en argument aux verrous.

♣. On omet l'évaluation paresseuse de la conjonction pour simplifier l'écriture de la preuve.

```
Procedure create(n) :
    Soit Ticket un tableau de n entiers initialisés à 0
    retourner Ticket

Procedure lock(Ticket, i) :
1   Ticket[i] ← 1 + max{Ticket[j] | j ∈ ⟦0, n − 1⟧}
2   pour j = 0 à N − 1 faire                          // Pour chaque autre fil d'exécution
        //On attend qu'il ait un moins bon ticket que nous, ou plus de ticket du tout
3       tant que Ticket[j] ≠ 0 et Ticket[j] < Ticket[i] faire Rien

Procedure unlock(Ticket, i) :
4   Ticket[i] ← 0
```

La ligne $\mathsf{Ticket}[i] \leftarrow 1+\max\{\mathsf{Ticket}[j] \mid j \in [\![0, n-1]\!]\}$ de l'algorithme n'est pas effectuée de manière atomique, ce qui peut conduire plusieurs fils d'exécution à avoir le même numéro de ticket. Afin de pallier ce problème on oblige les fils d'exécution à attendre que chaque autre fil d'exécution intéressé par la section critique ait fini de calculer le $\max$.

```
Procedure create(n) :
    Soit Ticket un tableau de n entiers initialisés à 0
    Soit EnCalcul un tableau de n booléens initialisés à F
    retourner (Ticket, EnCalcul)

Procedure lock(Ticket, i) :
1   EnCalcul[i] ← V
2   Ticket[i] ← 1 + max{Ticket[j] | j ∈ ⟦0, n − 1⟧}
3   EnCalcul[i] ← F
4   pour j = 0 à n − 1 faire                          // Pour chaque autre fil d'exécution
        //On attend qu'il ait fini le calcul du max
5       tant que EnCalcul[j] faire Rien
        //Puis on attend qu'il ait un moins bon ticket que nous, ou plus de ticket du tout
6       tant que Ticket[j] ≠ 0 et Ticket[j] < Ticket[i] faire Rien

Procedure unlock(Ticket, i) :
7   Ticket[i] ← 0
```

Cette version de l'algorithme a toutefois un dernier inconvénient : deux fils d'exécution a et b peuvent avoir le même numéro de Ticket de part le calcul concurrent des max. Prenons par exemple le cas où deux fils d'exécution calculent de manière concurrente le maximum alors qu'ils ont tout deux une valeur de ticket de 0, ils lisent chacun la valeur 0 de l'autre fil d'exécution, puis choisissent la valeur de ticket 1. Afin de pallier ce problème, on autorise différents fils d'exécution à avoir le même numéro de ticket, en cas d'égalité c'est le fil d'exécution de plus petit numéro qui rentre en premier dans la section critique. Ainsi les fils d'exécution ne sont plus comparés seulement par ticket, mais par la relation d'ordre lexicographique sur la valeur du ticket puis la valeur de leur identifiant. On note $\preceq_l$ cette relation d'ordre lexicographique.

Procedure create(n) :
    Ticket est un tableau de n entiers initialisés à 0.
    EnCalcul est un tableau de n booléens initialisés à F.
    retourner (Ticket, EnCalcul)

Procedure lock(Ticket,EnCalcul, i) :
1 EnCalcul$[i] \leftarrow$ V
2 Ticket$[i] \leftarrow 1 + \max\{\text{Ticket}[j] \mid j \in [\![0, n-1]\!]\}$
3 EnCalcul$[i] \leftarrow$ F
4 **pour** $j = 0$ **à** $n - 1$ **faire** // Pour chaque autre fil d'exécution
        //On attend qu'il ait fini le calcul du max
5 **tant que** EnCalcul$[j]$ **faire** Rien
        //Puis on attend qu'il ait un moins bon ticket que nous, ou plus de ticket du tout
6 **tant que** Ticket$[j] \neq 0$ et $((\text{Ticket}[j], j) \prec_l (\text{Ticket}[i], i))$ **faire** Rien

Procedure unlock(Ticket, i) :
7 Ticket$[i] \leftarrow 0$

Algorithme 9 – Algorithme de la boulangerie de Lamport

4.4 En OCaml, en C

4.4.1 En OCaml

Le type abstrait Verrou est implémenté en OCaml par le module `Mutex`. Les verrous sont des objets de type `Mutex.t`. On les manipule à travers les trois opérations suivantes :
- la fonction `Mutex.create : unit -> Mutex.t` qui crée un verrou ;
- la fonction `Mutex.lock : Mutex.t -> unit` qui permet de verrouiller un verrou ;
- la fonction `Mutex.unlock : Mutex.t -> unit` qui permet de déverrouiller un verrou.

Une fois un verrou `v` créé, on garantit l'exclusion mutuelle entre les sections de codes délimitées par un appel à `(Mutex.lock v)` et un appel à `(Mutex.unlock v)`.

Exemple 4.3
On donne ci-dessous le cas de deux fils d'exécution devant réaliser la même tâche, à savoir incrémenter un compteur partagé un nombre donné de fois.

```
type args =
  { nbt : int ; mutable cpt : int }
let verrou = Mutex.create ()
(** Ajoute [a.nbt] fois 1 à [a.cpt] *)
let add_one (a : args) : unit =
  for i = 1 to a.nbt do
    Mutex.lock verrou;
    a.cpt <- a.cpt + 1;
    Mutex.unlock verrou
  done
```

```

let main (n : int) : unit =
  let a = {nbt = n; cpt = 0} in
  let f1 = Thread.create add_one a in
  let f2 = Thread.create add_one a in
  Thread.join f1;
  Thread.join f2;
  let res = a.cpt in
  assert (res = 2 * n)
```

4.4.2 En C

Le type abstrait Verrou est implémenté en C par le type `pthread_mutex_t` de la bibliothèque `pthread`. Une fois un verrou `v` déclaré et initialisé grâce à un appel à `pthread_mutex_init(&v, NULL)`, on garantit l'exclusion mutuelle entre les sections de codes délimitées par un appel à `pthread_mutex_lock(&v)`

et un appel à `pthread_mutex_unlock(&v)`.

Exemple 4.4
On donne ci-dessous une exemple avec trois fils d'exécution devant réaliser la même tâche, à savoir incrémenter un compteur partagé un nombre donné de fois. On définit donc une seule tâche (la fonction `add_one`), et une seule structure pour passer à cette fonction ses arguments (la structure `args`). Afin de mettre en exclusion mutuelle les incrémentations du compteur faites par chaque fil réalisant `add_one`, on protège la ligne 16 grâce à un verrou déclaré au préalable (ligne 5).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  pthread_mutex_t verrou ;
6  typedef struct args_s {
7    int* cpt;          /* adresse compteur */
8    int nbt;       /* nb de tours souhaités */
9  } args ;
10
11 /* incr. arg->nbt fois *(arg->cpt) */
12 void* add_one(void* arg) {
13   args* a = (args*) arg;
14   for (int i = 0; i < a->nbt ; i++) {
15     pthread_mutex_lock(&verrou);
16     *(a->cpt)= *(a->cpt) + 1;
17     pthread_mutex_unlock(&verrou);
18   }
19   return NULL;
20 }
```

```
21
22 int main(int argc, char* argv[]) {
23
24   int cpt = 0; // variable partagée
25   args a1 = {.cpt = &cpt, .nbt = 10000};
26
27   pthread_mutex_init(&verrou, NULL);
28
29   pthread_t p1, p2, p3;
30   pthread_create(&p1, NULL, add_one, &a1);
31   pthread_create(&p2, NULL, add_one, &a1);
32   pthread_create(&p3, NULL, add_one, &a1);
33
34   pthread_join(p1, NULL);
35   pthread_join(p2, NULL);
36   pthread_join(p3, NULL);
37
38   return 0;
39 }
```

5 Sémaphores

5.1 Définition

Les sémaphores sont une généralisation des verrous.

Métaphore explicative. Il y a un nombre n de postes disponibles dans une salle informatique. Des étudiants souhaitent utiliser ces postes, mais sont en nombre supérieur au nombre de poste. On installe donc un surveillant qui est en charge de laisser entrer les étudiants dans la salle informatique. Ce surveillant ne regarde jamais dans la salle, il maintient seulement un compteur du nombre de postes disponibles dans la salle, ainsi qu'un registre des étudiants souhaitant accéder à la salle informatique. Lorsqu'un étudiant arrive, il y a deux cas :

- ou bien le nombre de places disponibles dans la salle est non nul, auquel cas le surveillant laisse entrer l'étudiant et décrémente le compteur du nombre de places disponibles dans la salle ;
- ou bien le nombre de places disponibles dans la salle est nul, auquel cas le surveillant demande à l'étudiant de patienter, et il l'inscrit sur le registre des étudiants en attente.

Lorsqu'un étudiant sort de la salle, il y a deux cas :

- ou bien il n'y a pas d'étudiants attendant l'accès à la salle, auquel cas le surveillant incrémente le nombre de places disponibles dans la salle ;
- ou bien il y a des étudiants qui attendent de pouvoir accéder à la salle, le surveillant va alors chercher un des étudiants de son registre et lui donne accès à la salle.

Définition 5.1

Le type de donnée abstrait Semaphore fournit une définition de type semaphore et trois fonctions de manipulation de ce type :

- *create* : $\mathbb{N} \to$ *semaphore, une fonction de création d'une sémaphore pour un entier* n *;*
- *acquire* : *semaphore* $\to ()$*, une fonction d'aquisition du sémaphore ;*
- *release* : *semaphore* $\to ()$*, une fonction de libération du sémaphore.*

État du sémaphore L'état d'un sémaphore est représenté par un compteur et par la donnée d'un ensemble de fils d'exécution en attente.

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

Remarque 5.2

Contrairement à ce que la métaphore introductive pourrait laisser penser, on remarque que le type de donnée abstrait Semaphore ne fournit pas d'opération d'accès au nombre d'éléments en attente. De plus dans certains cas on pourra libérer le sémaphore avant d'y avoir accédé (*Cf.* section 5.2.3), ce qui, dans le cadre de la métaphore précédente permet de modéliser l'installation de machines supplémentaires dans la salle, ce qui augmente le nombre de places disponibles sans pour autant qu'un étudiant ait quitté la salle.

5.2 Quelques problèmes classiques de la concurrence

5.2.1 Le problème de l'exclusion mutuelle

On peut facilement garantir l'exclusion mutuelle à l'aide d'un sémaphore créé pour l'entier 1. Chaque fil d'exécution acquiert alors le sémaphore lorsqu'il entre dans sa section critique et il le libère lorsqu'il la quitte. Cette solution ne garantit pas l'absence de famine : lorsque plusieurs fils sont mis en attente, la sortie d'un fil d'exécution libère le sémaphore et l'un de ces fils en attente peut alors entrer en section critique, mais on ne sait pas lequel, ainsi rien n'empêche que l'un des fils reste toujours en attente tandis que d'autre répètent une infinité de fois leur section critique.

5.2.2 Problème *multiplex*

On considère la généralisation suivante du problème de l'exclusion mutuelle : N fils d'exécutions souhaitent accéder à une section critique, on autorise au plus $p \in [\![1, N]\!]$ fils d'exécution à être dans la section critique de manière simultanée. Ce problème se nomme le problème du **multiplex**. De même que dans le problème de l'exclusion mutuelle, les N fils d'exécution exécutent "en boucle" la suite d'instructions : Section non critique ; lock() ; Section critique ; unlock().
Le problème du *multiplex* peut être résolu au moyen d'un sémaphore en implémentant les fonctions lock et unlock de la manière suivante.

```
   Procedure create() :
1  |  retourner S Sémaphore initialisé par create(p)

   Procedure lock() :
2  |  acquire(S)

   Procedure unlock() :
3  |  release(S)
```

Algorithme 10 – Multiplex

Le système des N fils d'exécution assure alors l'invariant suivant : la valeur du sémaphore est de p moins le nombre de fils d'exécution se trouvant en section critique. En effet, cet invariant est initialement vrai et les fils d'exécutions décrémentent la valeur du sémaphore en entrant dans la section critique et l'incrémente en sortant, ce qui assure bien le maintient de l'invariant.

5.2.3 Le problème de la mise en séquence

Dans le cas de la programmation concurrente, le problème de la **mise en séquence** est celui d'assurer qu'une instruction d'un certain fil d'exécution est exécutée avant une certaine instruction d'un autre fil d'exécution. On peut imaginer par exemple que la seconde instruction dépend du résultat d'un calcul effectué par la première.
Ce problème peut être résolu au moyen d'un sémaphore. En effet considérons l'algorithme concurrent 11. L'instruction p_2 est nécessairement exécutée avant l'instruction q_2 : le fil d'exécution Q est bloqué par l'exécution du acquire(S) tant que P n'a pas exécuté l'instruction release(S), et *a fortiori*, l'instruction p_2.

$S \leftarrow$ Semaphore.create(0)	
Fil d'exécution P	Fil d'exécution Q
1 p_1 ; 2 p_2 ; 3 release(S) ; 4 p_3 ;	1 q_1 ; 2 acquire(S) ; 3 q_2 ; 4 q_3 ;

Algorithme 11 – Mise en séquence de p_2 et q_2 au moyen d'un sémaphore.

5.2.4 Producteurs-consommateurs

On considère une application dans laquelle des fils d'exécution produisent des ressources (les résultats d'un calcul par exemple) et stockent les résultats dans une mémoire bornée de taille $n \in \mathbb{N}$. On dit que ces fils d'exécution sont des **producteurs**. D'autres fils d'exécution consomment ces ressources, on dit que ce sont des **consommateurs**. Si les consommateurs consomment moins vite que les producteurs ne produisent, la mémoire servant de zone de stockage risque de "déborder". À l'inverse, si les consommateurs consomment plus vite que les producteurs ne produisent, la mémoire risque d'être vide. Le problème **producteur-consommateur** est donc de réguler la production et la consommation pour éviter un déséquilibre de la zone de stockage.

On propose le protocole suivant :

- un fil d'exécution producteur est mis en attente s'il souhaite écrire dans une mémoire pleine ;
- un fil d'exécution consommateur est mis en attente s'il souhaite écrire dans une mémoire vide ;
- un seul fil d'exécution est en train d'accéder à la mémoire (lecture ou écriture) à la fois.

On propose pour cela l'implémentation utilisant les structures suivantes :
- un verrou Acces, protège l'accès à la lecture et/ou à l'écriture dans la mémoire ;
- un sémaphore Vide, interdisant l'accès à la lecture si la mémoire est vide, initialisé à la valeur 0 ;
- un sémaphore Plein interdisant l'accès à l'écriture si la mémoire est pleine, initialisé à la valeur n.

Le sémaphore Vide compte le nombre de places occupées dans la mémoire afin de mettre en attente les consommateurs lorsque la mémoire est vide, tandis que le sémaphore Plein compte le nombre de places libres dans la mémoire de mettre en attente les producteurs lorsque la mémoire est pleine.

Algorithme des consommateurs	Algorithme des producteurs
1 acquire(Vide) ; 2 lock(Acces) ; 3 Lecture ; 4 unlock(Acces) ; 5 release(Plein) ; 6 Traitement de la donnée lue	1 Génération d'une donnée ; 2 acquire(Plein) ; 3 lock(Acces) ; 4 Écriture ; 5 unlock(Acces) ; 6 release(Vide) ;

Ainsi les consommateurs diminuent le nombre de places occupées et augmentent le nombre de places libres, c'est l'inverse pour les producteurs.