
Chapitre 8 : Concurrency

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 ;	1 q_1 ;
2 p_2 ;	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 : $q_1 \rightarrow q_2 \rightarrow p_1 \rightarrow q_3 \rightarrow p_2$. 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 :

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

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 bien 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.

$n \leftarrow$ 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

♣. On se limite aux états accessibles

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

- Deux entiers : les pointeurs d'instructions, (\bullet 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.

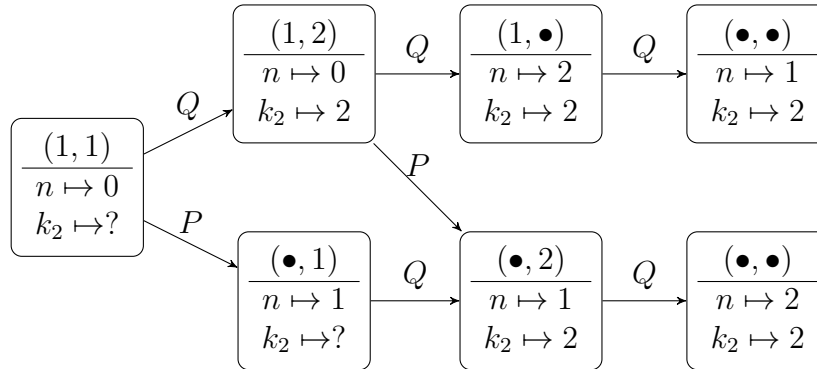


FIGURE 1 – Diagramme d'états de l'algorithme 2

2 Atomicité

Afin de rendre plus clairs les algorithmes, on se permet généralement d'utiliser des instructions élémentaires "de haut niveau" \clubsuit . 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'incrément $c++$.

Exemple de l'incrément. L'instruction d'incrément $c++$ 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>	1	Registre $\leftarrow c$;
add	\$0x1,%eax	2	Registre $++$;
mov	%eax,0x2d6f(%rip) # 4070 <c>	3	$c \leftarrow$ Registre;

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

(b) Équivalent en pseudo-code

Considérons les deux algorithmes concurrents ci-dessous.

c \leftarrow variable globale initialisée à 0;	
Fil d'exécution P	Fil d'exécution Q
1 $c++$;	1 $c++$;

Algorithme 3 – Double incrément de c : version haut niveau

\clubsuit . 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

$c \leftarrow$ variable globale initialisée à 0;	
Fil d'exécution P	Fil d'exécution Q
Soit regP var. loc. ; P_1 $\text{RegP} \leftarrow c$; P_2 $\text{RegP}++$; P_3 $c \leftarrow \text{RegP}$;	Soit regQ var. loc. ; Q_1 $\text{RegQ} \leftarrow c$; Q_2 $\text{RegQ}++$; Q_3 $c \leftarrow \text{RegQ}$;

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 $P_1, Q_1, P_2, Q_2, P_3, Q_3$, 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 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*`.

♣. *thread* en anglais.

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.

```
1  (* Déclaration d'un type structuré permettant le stockage des arguments
2  (nb) et de la valeur de retour (res) de la fonction au_carre. *)
3  type args =
4  {
5  nb: int          ;
6  mutable res : int ;
7  }
8
9  let au_carre (args: args): unit =
10  (* On écrit le résultat du calcul dans de la mémoire accessible par la
11  fonction appelante. *)
12  args.res <- args.nb * args.nb
13
14  let () =
15  let arg1 = {nb = 2; res = -1} in
16  let arg2 = {nb = 9; res = -1} in
17  (* On "lance" le thread pb : il doit exécuter la fonction au_carre sur
18  l'argument 2 et écrire le résultat dans le champs res de arg1. *)
19  let pa = Thread.create au_carre arg1 in
20
21  (* On "lance" le thread pb : il doit exécuter la fonction au_carre sur
22  l'argument 9 et écrire le résultat dans le champs res de arg2. *)
23  let pb = Thread.create au_carre arg2 in
24
25  (* On attend que les deux exécutions soient terminées. *)
26  Thread.join pa;
27  Thread.join pb;
28  assert (arg1.res = 4 && arg2.res = 81)
```

Mise au carré en C.

```
1  #include <pthread.h>
2  #include <assert.h>
3
4  /* Déclaration d'un type structuré permettant le stockage des arguments
5  * (nb) et de la valeur de retour (res) de la fonction au_carre. */
6  struct args_s {
7  int nb;
8  int* res;
9  };
10 typedef struct args_s arg_carre;
11
12 /* La fonction que l'on souhaite exécuter de manière concurrente. */
13 void* au_carre(void* args) {
```

```

14  /* On transtype l'argument de type void* qui est un arg_carre* */
15  arg_carre* a = (arg_carre*) args;
16  /* On écrit le résultat du calcul dans de la mémoire accessible par la
17   * fonction appelante. */
18  *(a->res) = a->nb * a->nb;
19  return NULL;
20 }
21
22 int main(){
23     int resA, resB;
24     arg_carre argsA = {2, &resA};
25     arg_carre argsB = {9, &resB};
26     /* Déclaration des deux threads */
27     pthread_t pA, pB;
28     /* On "lance" le thread pA : il doit exécuter la fonction au_carre sur
29      * l'argument 2 et écrire la valeur résultat dans resA. */
30     pthread_create(&pA, NULL, au_carre, &argsA);
31
32     /* On "lance" le thread pB : il doit exécuter la fonction au_carre sur
33      * l'argument 9 et écrire la valeur résultat dans resB. */
34     pthread_create(&pB, NULL, au_carre, &argsB);
35
36     /* On attend que les deux exécutions soient terminées. */
37     pthread_join(pA, NULL);
38     pthread_join(pB, NULL);
39     assert((resA == 4) && (resB == 81));
40     return 0;
41 }

```

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 ; <u>Pré-section critique ;</u> Section critique ; <u>Post-section critique ;</u> Section non critique ;	tant que ... faire Section non critique ; <u>Pré-section critique ;</u> Section critique ; <u>Post-section critique ;</u> 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 : () → verrou, qui correspond à la création de variables globales utiles au protocole ;*
- *lock : verrou → (), qui correspond à la partie du protocole précédant la section critique ;*
- *unlock : verrou → (), 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.

$V \leftarrow \text{create}();$	
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.

Procédure create() : Soit Dedans un tableau de deux booléens initialisés à F retourner Dedans
Procédure lock(Dedans, i) : $o \leftarrow 1 - i$ // L'autre 1 tant que Dedans[o] faire Rien // On attend tant que l'autre est en section critique 2 Dedans[i] $\leftarrow V$ // On se signale comme étant en section critique
Procédure unlock(Dedans, i) : 3 Dedans[i] $\leftarrow F$ // 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 $P_1; Q_1; P_2; Q_2; P_4; Q_4$) ce qui correspond au cas où les deux fils d'exécution sont en section critique. Ainsi le verrou ne vérifie pas la propriété 1.

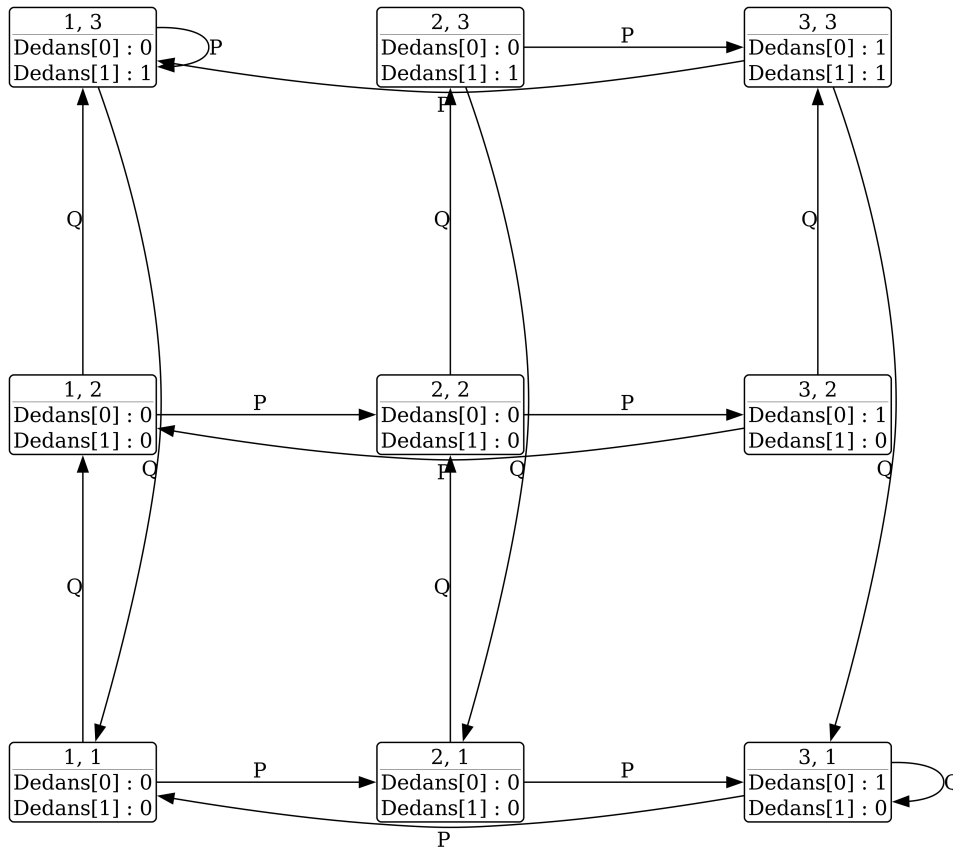


FIGURE 3 – Diagramme d'états de la première version

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.

```

Procédure create() :
  | Soit Want un tableau de deux booléens initialisés à F
  | retourner Want

Procédure lock(Want, i) :
1 |  $o \leftarrow 1 - i$  // L'autre
2 | Want[i] ← V // On dit vouloir aller en section critique
3 | tant que Want[o] faire Rien // On attend tant que l'autre veut aller en section critique

Procédure unlock(Want, i) :
4 | 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 ; ...

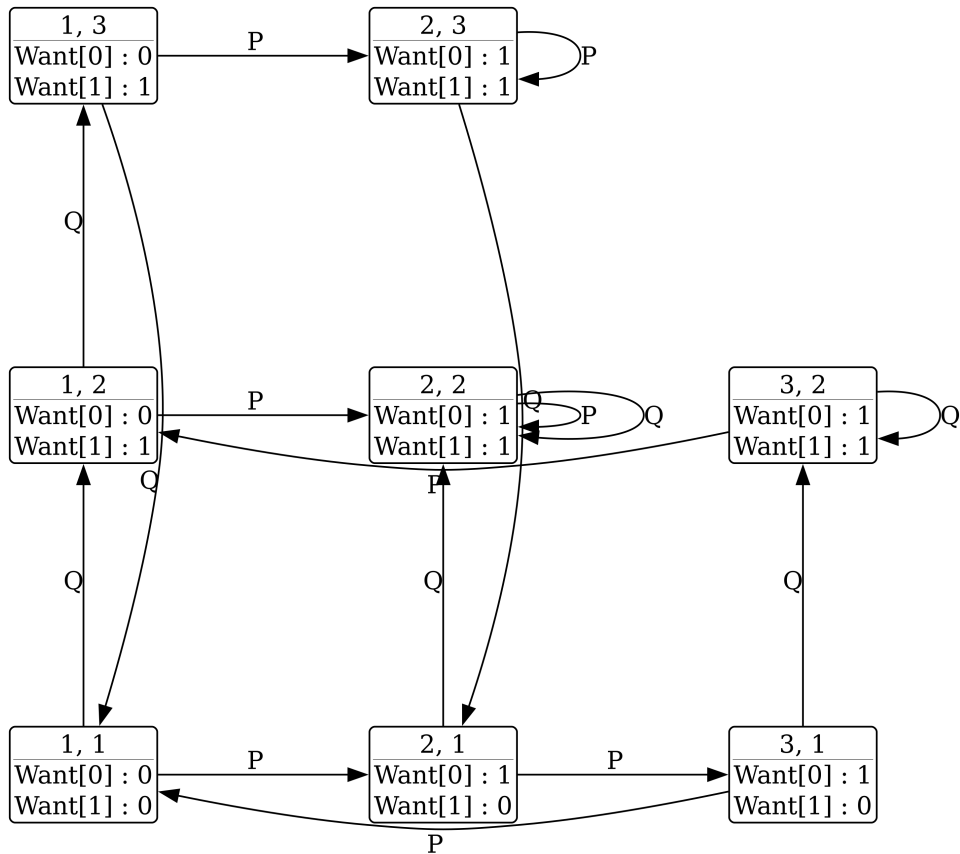


FIGURE 4 – Diagramme d'états de la seconde version

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.

```

Procédure create() :
  Soit  $Turn$  une variable entière initialisée à 0
  retourner  $Turn$ 

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

Procédure unlock( $Turn, i$ ) :
  2  $o \leftarrow 1 - i$  // L'autre
   $Turn \leftarrow o$  // 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.

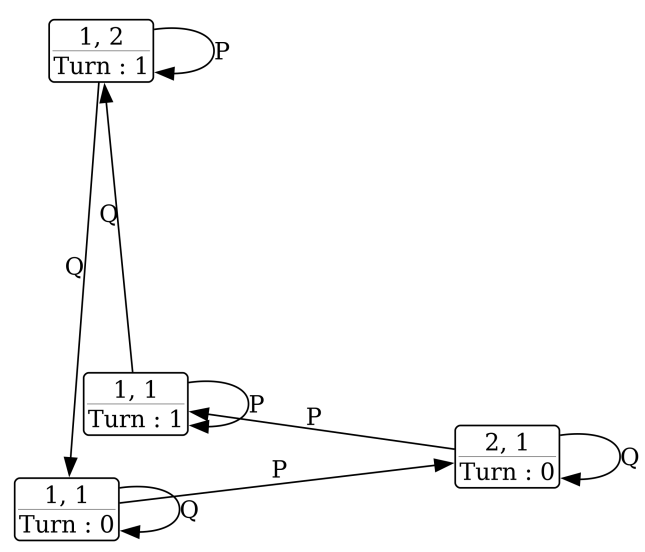


FIGURE 5 – Diagramme d'états de la troisième version

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.

```

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

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

Procédure 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.

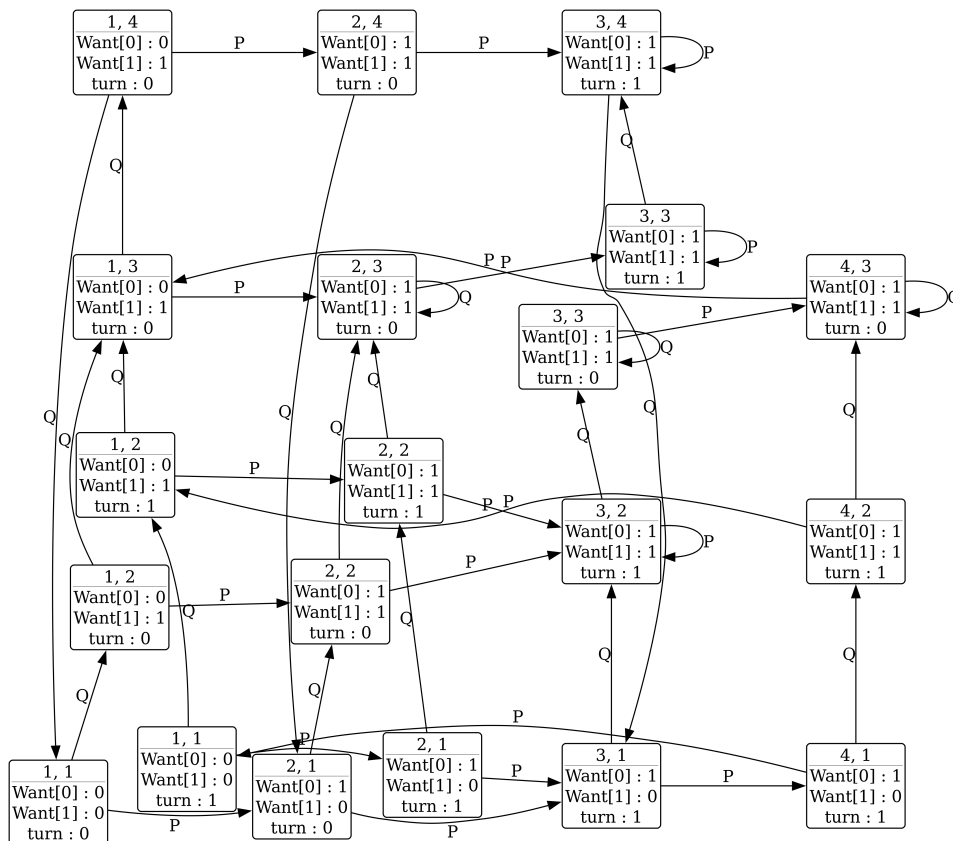


FIGURE 6 – Diagramme d'états de l'algorithme de Peterson

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 :