Chapitre 10 : Parallélisation, concurrence et synchronisation

21 novembre

1 Rappels sur les processus

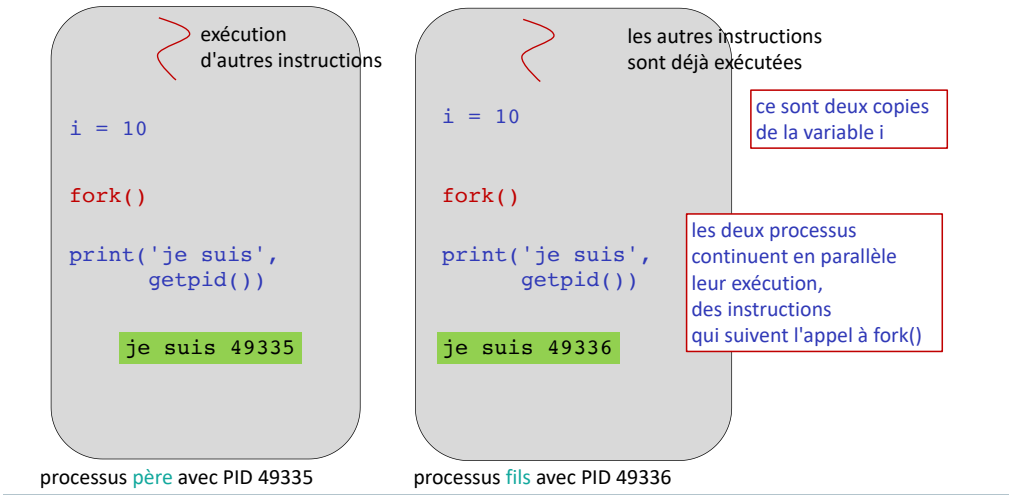
Le système d'exploitation nous fournit une vision virtuelle des ressources matérielles offertes par la machine par l'intermédiaire (entre autres) des processus, des flux d'entrée-sortie et de la mémoire virtuelle.

Un processus est un programme en cours d'exécution. C'est la ressource virtuelle que le système d'exploitation nous fournit pour se substituer au processeur (l'unité matérielle de calcul). Un processus dispose de son propre espace mémoire séparé qui lui donne l'illusion d'être le seul à utiliser cette dernière.

On rappelle que l'utilisateur écrit du code dans un langage haut niveau qui est enregistré dans un fichier et donc stocké sur le disque, ce code source est une donnée statique. Ce code est ensuite traduit en fichier exécutable qui, une fois qu'il est lancé, donne lieu à ce qu'on appelle un processus c'est-à-dire un programme en cours d'exécution. Un processus est une donnée dynamique.

L'ordonnanceur est l'outil du système d'exploitation qui permet d'organiser les accès des différents processus au processeur. Il donne accès à l'espace de calcul à un processus puis le met en attente afin de donner accès à un autre processus etc...

Un nouveau processus est automatiquement créé dès que l'on lance un exécutable. Une autre manière de créer un processus est d'utiliser l'appel système `fork` qui permet à un processus de se dupliquer afin de donner lieu à un nouveau processus qui en est une copie identique. Ce nouveau processus dispose de son propre espace mémoire indépendant. Le nouveau processus se retrouve exactement dans le même état que le processus père qui l'a créé, les instructions qui précèdent la commande `fork` sont donc déjà exécutées et il devra exécuter celles qui suivent l'instruction `fork`.



2 Fils d'exécutions

Les processus dont nous venons de parler sont parfois appelés processus lourds car ils utilisent un espace mémoire dédié et sont gérés par le système d'exploitation via une structure de donnée qui contient toutes les informations le concernant. Ce type de processus nécessite donc beaucoup de ressources et ses opérations courantes (création, ordonnancement, changement d'état, terminaison) sont lentes ce qui ralentit le fonctionnement global du système.

De plus, les processus sont isolés, il est envisageable mais compliqué et couteux de les faire "communiquer".

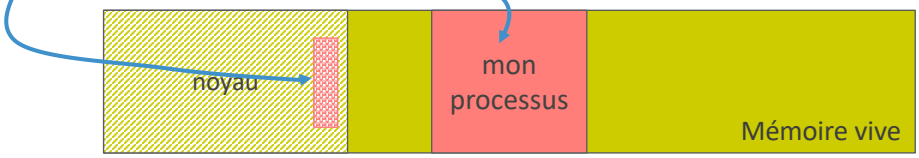
vision UTILISATEUR

Un processus représente l'exécution d'un programme

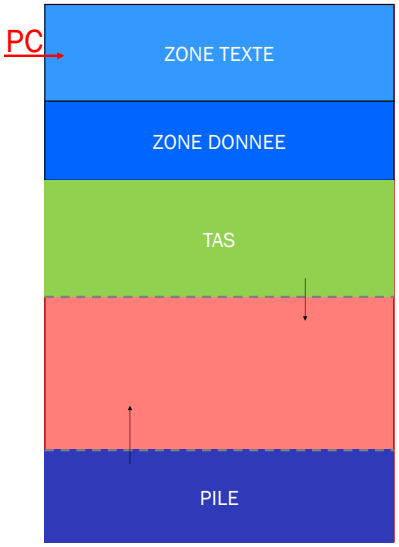
vision SYSTEME

Un processus est la somme de

- Un espace mémoire dédié = le processus est chargé en mémoire
- Une structure de gestion maintenue par le système où il y a **toutes** les informations le concernant



La figure suivante rappelle l'organisation mémoire d'un processus vue en première année :



Le code =
les instructions exécuter

Les données statiques
(connues en avance)

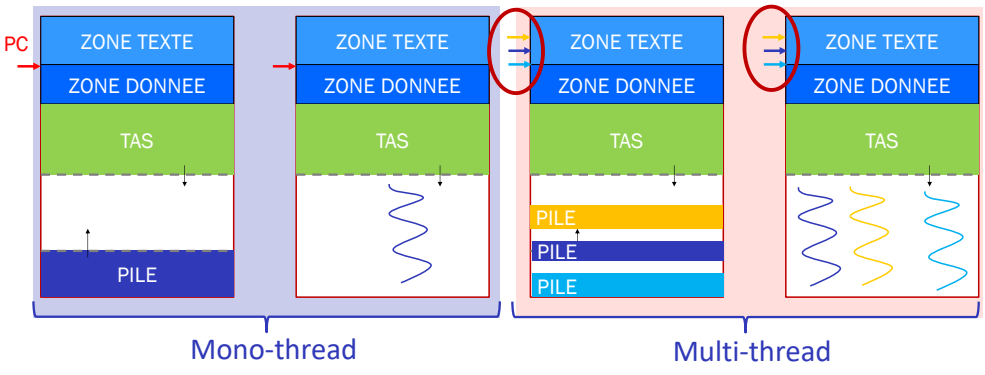
Les données dynamiques
(apparaissent en cours d'exécution)

PC = Program Counter Register
pointe vers la prochaine instruction

L'historique d'exécution
(dans quelle fonction je suis,
d'où je viens)

Nous allons maintenant parler de processus légers aussi appelés fils d'exécution ou threads en anglais. Les avantages des threads sont leur efficacité et le fait qu'ils pourront se coordonner. Leur gros inconvénient est le fait qu'ils vont partager leur mémoire ce qui induit un risque majeur. En effet, si un thread modifie l'état de la mémoire au moment où un autre thread est en train de la lire alors les résultats pourront être faussés. Nous aurons donc besoin d'outils qui permettront de résoudre ces problèmes.

Lorsque l'on dispose de deux processus d'un même programme, on ne peut pas factoriser le contenu du code ni partager les données et on doit donc travailler avec deux mémoires dupliquées. Les processus sont isolés. La parallélisation de l'exécution des processus implique donc d'utiliser une grande quantité de mémoire. Les threads vont être contenus à l'intérieur d'un même processus et vont permettre d'être exécutés indépendamment comme le seraient des processus distincts tout en partageant de l'information qui sera alors sauvegardée dans le processus qui les contient. Au sein d'un même processus, les différents threads partagent le même code et la même mémoire dynamique (le tas).



On retiendra que :

1. Les threads sont des entités d'exécution plus légères que les processus.
2. Ils évoluent au sein des processus
3. Ils partagent les ressources fournies par les processus
4. La programmation concurrente avec threads est difficile !

3 Non déterminisme

Nous allons ici introduire la syntaxe relative aux thread et regarder de premiers exemples.
En C, on utilisera la librairie `pthread.h`.
En OCAML, on utilisera le type `Thread.t` de la librairie `Thread`.

3.1 Un premier exemple de création de thread en OCAML

```
let f n =
   print_string(n);
   print_string("\n");;

 print_string("debut");;
 let t1 = Thread.create f "thread 1";;
 let t2 = Thread.create f "thread 2";;
 Thread.join t1;;
 Thread.join t2;;
 print_string("fin");;
```

3.2 Non déterminisme de l'exécution.

Exécutons le programme précédent à plusieurs reprises. Nous observons que l'ordre des appels n'est pas toujours le même. En effet, une fois les thread créés, l'ordonnancement de leurs exécutions respectives est non déterministe.

3.3 Le même exemple en C :

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

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

int main(void* arg){
    pthread_t p1, p2;
    printf("main begins\n");

    pthread_create(&p1, NULL, mythread, "thread 1" );
    pthread_create(&p2, NULL, mythread, "thread 2");

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

    printf("main ends \n");
    return 0;


}
```

3.4 Syntaxe :

Dans chacun des deux langages, on définit la fonction sur laquelle on lancera les threads puis on crée le thread et enfin on utilise une fonction `join` qui suspend l'exécution du thread appelant (votre programme principal) jusqu'à ce que le thread en argument soit terminé.

En OCAML, on utilise le type `Thread.t` et en C le type `pthread_t`.

En OCAML, la fonction `Thread.create` prend en entrée la fonction accompagnée de son entrée et la fonction `Thread.join` prend en entrée le thread.

En C, le passage d'une fonction en argument étant quelque chose de subtil les fonctions `pthread_create` et `pthread_join` ont des prototypes un peu compliqués :

```
pthread_create(pthread_t* thread, pthread_attr_t* attr, void* (*f)(void*),void* arg)
```

et

```
pthread_join(pthread_t thread, void** ret)
```

Pour l'utilisation de `create` : le premier argument est l'adresse d'un thread préalablement déclaré, le deuxième argument correspond à des options auxquelles nous ne nous intéresserons pas et donc cet argument vaudra systématiquement `NULL` pour nous, le troisième argument est la fonction qui doit être exécutée par le thread et le dernier argument est l'argument avec lequel appeler cette fonction.

Avant de créer un thread, on écrira donc une fonction souvent appelée `mythread` de prototype :

```
(void*) mythread (void* in)
```

On remarquera que l'entrée et la sortie étant de type `void*` on fera des cast avec les entrées et sorties réelles. On remarquera de plus que la fonction ne peut prendre qu'un seul argument en entrée, si on veut simuler des fonctions avec plusieurs arguments il faudra créer une structure contenant tous les arguments souhaités et la fonction `mythread` prendra en entrée un pointeur sur une telle structure.

La fonction `join` prend en entrée le thread et une adresse à laquelle récupérer la valeur calculée par la fonction. Le programme stipule qu'on ne doit pas se servir de cela et limiter l'utilisation à un second paramètre qui vaut `NULL`. Ceci implique de travailler avec des variables globales à modifier si on veut récupérer les résultats des threads.

Un dernier point important : pour utiliser les thread on doit utiliser des options de compilation : en OCAML on pourra compiler avec la commande suivante :

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

En C, on ajoutera le flag `-pthread` dans les options.

3.5 Entrelacement

Une fois les threads créés, l'ordonnancement de leurs actions est non déterministe : l'un des thread prend la main puis se met en pause pour que le second travaille et ainsi de suite avec des changements qui obéissent aux règles édictées par l'ordonnanceur qui sont non déterministes.

On retiendra donc la définition suivante :

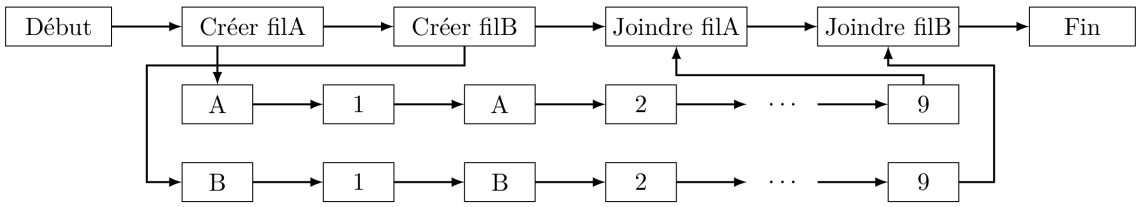
Définition : Un algorithme ou programme est dit séquentiel si les instructions qui le composent sont exécutées l'une après l'autre, toujours dans le même ordre. Il est dit concurrent s'il contient plusieurs unités, qu'on appelle fil d'exécution ou thread, chaque fil s'exécutant séquentiellement.

Dans un programme concurrent, il n'y a donc pas la garantie que les instructions seront exécutées toujours dans le même ordre.

Reprenons le premier exemple et étoffons le un tout petit peu pour mettre cela en évidence :

```
let limite = 9;;
let f n =
   for i=0 to limite do
      print_string(n);
      print_int(i);
      print_string("\n");
   done;;

 print_string("debut");;
 let t1 = Thread.create f "thread A";;
 let t2 = Thread.create f "thread B";;
 Thread.join t1;;
 Thread.join t2;;
 print_string("fin");;
```



La seule contrainte sur l'exécution du programme est d'exécuter les instructions dans un ordre topologique du graphe. Cependant, comme il existe plusieurs ordres topologiques, plusieurs exécutions sont possibles. On peut imaginer que tous les affichages suivants sont possibles.

```
A1A2A3A4A5A6A7A8A9B1B2B3B4B5B6B7B8B9
AB11BA2A23B3BA44BA55B6AB76AB7A88B9A9
B1BA2B13BA24A3A4A5AB5B6A7A68B7A9B8B9
```

De même, il est tout à fait possible d'imaginer que le fil A commence ses instructions après avoir fini toutes les instructions du fil B.

Ainsi, l'exécution d'un programme concurrent est **non-déterministe** : il n'y a pas qu'une seule exécution possible.

3.6 Exemple 1 : Somme des éléments d'un tableau

Ecrire une fonction qui calcule la somme des éléments d'un tableau passé en entrée.
Ecrire une fonction qui calcule ce résultat en utilisant 2 thread (le faire en C et en OCAML).
Ecrire une fonction qui le fait avec n thread (en Tp, en C et en OCAML).

3.7 Exemple 2 : produit matriciel

Ecrire, en OCAML, un programme qui fait le produit de deux matrices en utilisant un thread pour le calcul de chaque case.

4 Les problèmes de concurrence

Nous allons maintenant mettre en évidence les problèmes de concurrence qu'induisent l'utilisation des threads.

4.1 Des résultats étonnants

Exécutons maintenant le code suivant :

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

```
#include <pthread.h>

static int counter =0;

void* mythread (void* arg){
    printf("%s : begin\n", (char*)arg);
    for (int i=0;i<9999;i++){
        counter++;
    }
    printf("%s : done\n", (char*)arg);
    return NULL;
}

int main(void* arg){
    pthread_t p1, p2;
    printf("main begins, counter = %d\n",counter);

    pthread_create(&p1, NULL, mythread,"thread 1" );
    pthread_create(&p2,NULL,mythread,"thread 2");

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

    printf("main ends, counter = %d \n", counter);
    return 0;


}
```

Comment expliquer le phénomène ?

Voici le code assembleur de `counter++` :

```
mov counter, %eax
addl $1, %eax
movl %eax, counter
```

Ainsi, suivant l'entrelacement obtenu lors de l'exécution, les résultats attendus ne sont pas forcément corrects.

Exercice 1 : Déterminer les valeurs minimales et maximales que peut prendre `n` à la fin de l'exécution des deux threads.

4.2 Atomicité

Une opération atomique est une opération qui ne peut pas être interrompue. Un thread en train d'effectuer cette opération ne peut pas passer la main tant qu'elle n'est pas terminée. Seules les instructions machines sont atomiques par défaut. Nous allons tout de même voir par la suite comment rendre des instructions atomiques.

4.3 Sections critiques

Une section critique est une portion de code qui, pour garantir la sûreté du résultat, ne peut être exécutée simultanément que par un nombre maximal de threads différents (en pratique nous nous restreindrons à un seul thread).

Par exemple, l'instruction `counter++` du code précédent est une section critique.

Plus la section critique est large, plus la sureté est garantie, mais une grande section critique empêche les autres threads de s'exécuter et donc ralentit l'exécution globale, il est donc nécessaire d'avoir les sections critiques les plus petites possibles (contenant le moins d'instructions possibles) mais garantissant l'absence de concurrence.

Exercice 2 : On considère une variable n initialisée à 0 et trois fils d'exécutions qui effectuent les opérations suivantes :

— T_1 : $a \leftarrow n$, $n \leftarrow 1$, $b \leftarrow n$;
— T_2 : $c \leftarrow n$, $n \leftarrow 2$, $d \leftarrow n$;
— T_3 : $e \leftarrow n$, $f \leftarrow n$.

On suppose que l'instruction $x \leftarrow y$ consiste à écrire le contenu de y dans x et est une instruction atomique. Après l'exécution des trois fils :

1. que peut valoir c ?
2. que peut valoir b ?
3. que peut valoir e ?
4. si c vaut 1, que peut valoir d ?
5. si f vaut 0, que peut valoir e ?
6. si a vaut 2 et d vaut 1, que peut valoir b ?

Exercice 3 : On considère le code suivant :

```
int k = 0, i = 0;

void* foo(void* arg){
    while (i < 10){
        i++;
        k++;
    }
    return NULL;
}

int main(void){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, foo, NULL);
    pthread_create(&t2, NULL, foo, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Quelles sont les valeurs minimale et maximale que peut prendre théoriquement `k` à la fin de l'exécution ?

5 SYNCHRONISATION

5.1 VERROUS, MUTEX

Pour réaliser une section critique on utilise une primitive appelée verrou ou `mutex`. Un verrou est une variable qui dispose de deux opérations : `lock` et `unlock`. Un seul thread peut disposer du verrou. Pour délimiter une section critique, on utilise donc le principe suivant :

```
lock(m)
<section critique>
unlock(m)
```

Ainsi, lorsqu'un thread entre dans la section critique, le verrou est bloqué et aucun autre thread ne peut rentrer dans la même section de code avant qu'il ne l'est déverrouillé.

Il existe plusieurs manières d'implémenter en pratique les verrous mais chacune d'entre elle doit garantir les points suivants :

1. un seul thread peut être dans la section critique (exclusion mutuelle)
2. un thread ne peut pas attendre indéfiniment pour entrer en section critique (absence de famine).
3. un thread en dehors de la section critique ne peut pas bloquer les autres threads (pas d'interblocage).

Voici la syntaxe que l'on pourra utiliser :

1. en OCAML : on utilise le type `Mutex.t`
   (a) `Mutex.create : unit -> Mutex.t` crée un nouveau mutex.
   (b) `Mutex.lock : Mutex.t->unit` vérouille le mutex passé en argument.
   (c) `Mutex.unlock : Mutex.t->unit` déverrouille le mutex passé en argument.
2. en C : on utilise le type `pthread_mutex_t`
   (a) pour initialiser un mutex lors de sa déclaration, on utilisera :
   `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER`
   (b) `pthread_mutex_lock` prend en argument l'adresse d'un mutex et le verrouille.
   (c) `pthread_mutex_unlock` prend en argument l'adresse d'un mutex et le déverrouille.

L'idée fondamentale est qu'un seul thread à la fois peut verrouiller le mutex, ainsi si un thread essaie de verrouiller un verrou déjà verrouillé alors il est mis en attente. La fonction `unlock` réveille les threads éventuellement mis en attente qui seront alors en compétition pour acquérir le verrou.

5.2 EXEMPLE SOLUTIONNÉ

5.3 ALGORITHME DE PETERSON POUR IMPLÉMENTER UN MUTEX AVEC DEUX THREADS

La réalisation pratique d'un mutex est compliquée. Il existe néanmoins plusieurs solutions qui pour la plupart se basent sur des ressources matérielles.

Nous allons ici voir une solution algorithmique et donc purement logicielle qui nous permettra de comprendre les enjeux et les difficultés liés à la conception d'une telle primitive. Cet algorithme, inventé par Peterson en 1981, ne fonctionne que pour deux threads et recourt à l'attente active : lorsqu'un thread est bloqué avant d'accéder à la section critique, il doit continuer de travailler au sein d'une boucle plutôt que d'être mis en veille. Cette attente active est peu efficace.

Attention, on étudie ici des solutions d'un point de vue théorique, mais il est généralement fortement déconseillé de réimplémenter ces solutions soi-même : les mutex existent déjà en OCaml et en C, donc autant les utiliser.

1. Solution 1 : L'idée principale est d'utiliser un tableau de deux booléens indiquant pour chaque thread s'il est ou non en section critique :

```
bool flag[2]={false,false};
void lock(int i){
   int other = 1-i;
   while (flag[other]) ; //attente active
   flag[i]=true;
}

void unlock(int i){
   flag[i]=false;
}
```

   Quelle est le problème rencontré avec cette solution ?

2. Solution 2 : On modifie donc la sémantique du tableau qui contiendra maintenant `true` si les threads veulent ou non entrer en section critique :

```
bool want[2]={false,false};
void lock(int i){
   int other = 1-i;
   want[i]=true;
   while (want[other]) ; //attente active
}

void unlock(int i){
   want[i]=false;
}
```

   Le problème mis en évidence précédemment est-il solutionné ? Quel nouveau problème apparait ?

3. Solution 3 : On va du coup proposer l'utilisation d'une variable `turn` qui permet de déterminer lequel des deux threads peut acquérir le verrou :

```
int turn = 0;
void lock(int i){
   int other = 1-i;
   while (turn ==other) ; //attente active
}

void unlock(int i){
   turn = 1-i;
}
```

   Il y a encore un problème ici, l'identifier.

4. Solution **correcte** : La solution de Peterson combine l'utilisation du tableau `want` et de la variable `turn` :

```
int turn = 0;
bool want[2]={false,false};
void lock(int i){
   int other = 1-i;
   want[i]=true;
   turn = other;
   while (want[other] && turn == other) ; //attente active
}

void unlock(int i){
   want[i]=false;
}
```

Il reste à se convaincre que ceci satisfait bien les trois propriétés du mutex :

Un système de mutex doit nécessairement satisfaire l'exclusion mutuelle, très certainement l'absence d'interblocage et idéalement l'absence de famine.

5.4 Interblocages

Il faudra être très vigilant avec les mutex notamment lorsqu'on en utilise plusieurs car ils peuvent induire des situations d'interblocage où les deux threads se bloquent l'un l'autre comme l'illustre l'exemple suivant :

```
let m1 = Mutex.create();;
let m2 = Mutex.create();;

let f1()=
Mutex.lock m1;
Mutex.lock m2;
Mutex.unlock m2;
Mutex.unlock m1;;

let f2()=
Mutex.lock m2;
Mutex.lock m1;
Mutex.unlock m1;
Mutex.unlock m2;;

let t1 = Thread.create f1 ();;
let t2 = Thread.create f2 ();;
Thread.join t1;;
Thread.join t2;;
```

5.5 Algo de la boulangerie de Lamport : synchronisation d'un nombre quelconque de fils d'exécution

Nous allons maintenant voir un algorithme qui permet d'obtenir l'exclusion mutuelle d'un nombre quelconque de threads fixé. Cet algorithme, tout comme le précédent utilise l'attente active et n'a qu'un intérêt théorique et non pas pratique.

L'idée s'inspire de la gestion de l'attente dans un commerce ou une administration : chaque client (thread) recoit à son arrivée un ticket avec un numéro de passage et attend son tour (pour entrer en section critique). Lorsqu'il arrive un thread devra obtenir un numéro supérieur à tous les numéros déjà attribués auparavant afin de prévenir l'algorithme de la famine.

Tel quel, l'algorithme ne pourra pas garantir l'exclusion mutuelle car deux threads peuvent acquérir de manière concurentielle un même numéro et alors décider de rentrer ensemble dans la section critique.

L'algorithme va utiliser deux tableaux de taille n où n est le nombre total de threads. Chaque thread aura un identifiant i et le thread i ne pourra qu'écrire dans les cases i des tableaux. L'un des tableaux, noté `want` contiendra un booléen dans sa case i ssi le thread i souhaite entrer en section critique. L'autre tableau, noté `label`, contient en case i le numéro de passage obtenu par le thread i.

On peut donc considérer une première version insatisfaisante :

```
bool want[n];
int label[n];

void lock(int i){
   want[i]=true;
   //phase de récupération du numéro de passage
   int number = 0;
   for (int k=0;k<n;k++){
      if (number<label[k]){
         number=label[k];
      }
   }
   label[i]=number+1;
   //phase d'attente de son tour
   for (int k =0;k<n;k++){
      if (k !=i){
         while(want[k]&&label[k]<label[i]){
         }
      }
   }
}
```

```
void unlock(int i){
   want[i]=false;
}
```

Le problème de cette version, comme mentionné précédemment, est que l'exclusion mutuelle n'est pas garantie car deux exécutions concurrentes de la première phase peuvent aboutir à l'obtention d'un même ticket, on se prémunie de ce problème en utilisant l'identifiant des threads pour choisir parmi deux threads ayant le même numéro lequel est autorisé à entrer en section critique avant l'autre.

```
bool want[n];
int label[n];

void lock(int i){
   want[i]=true;
   //phase de récupération du numéro de passage
   int number = 0;
   for (int k=0;k<n;k++){
      if (number<label[k]){
         number=label[k];
      }
   }
   label[i]=number+1;
   //phase d'attente de son tour
   for (int k =0;k<n;k++){
      if (k !=i){
         while(want[k]&&(label[k]<label[i]||(label[k]==label[i]&&k<i))){
         }
      }
   }
}

void unlock(int i){
   want[i]=false;
}
```

Montrons que ce procédé respecte bien les propriétés attendues :

6 Sémaphores

L'exclusion mutuelle vue dans tout ce qui précède garantit qu'un thread seulement peut entrer en section critique.

Le sémaphore est une généralisation du mutex où on garantit qu'un nombre limité de thread peut entrer en section critique.

Le sémaphore est associé à deux fonctions : une fonction d'acquisition qui permet d'accéder à la section critique et une fonction qui permet à un thread de sortir de la section critique.

Fondamentalement, un sémaphore est un compteur (dont la valeur initiale est donnée lors de l'initialisation). Lorsqu'un thread utilise la fonction d'acquisition, si la valeur du compteur est strictement positive alors il se contente de le décrémenter et d'entrer en section critique sinon, si ce dernier est nul, alors il est mis en attente. Quand le thread sort de la section critique, le compteur est incrémenté et s'il y a des threads en attente alors l'un d'eux est réveillé.

6.1 Principe et syntaxe

1. En OCAML :
   (a) La fonction `Semaphore.Counting.make : int->Semaphore.Counting.t` crée un sémaphore dont la capacité est passée en entrée.
   (b) La fonction `Semaphore.Counting.acquire : Semaphore.Counting.t->unit` attend que le compteur devienne strictement positif puis le décrémente.
   (c) La fonction `Semaphore.Counting.release : Semaphore.Couting.t->unit` incrémente le compteur et réveille un thread en attente si besoin.
2. En C, on a les fonctions analogues :
   (a) La fonction `int sem_init(sem_t *sem, int pshared, unsigned int value)` initialise le sémaphore avec une capacité `v`. L'entier `pshared` spécifie si le sémaphore est ou non partagée : s'il vaut `0`, il l'est et ce sera toujours le cas pour nous.
   (b) La fonction `sem_wait(sem_t* s)` met en attente le thread appelant tant que le compteur est nul puis décrémente celui-ci.
   (c) La fonction `sem_post(sem_t* s)` incrémente le compteur et réveille un thread en attente.

6.2 Exemple 0 : Simulation de Mutex

Un sémaphore peut jouer le rôle d'un mutex, pour cela il suffit de choisir la capacité correctement.

```
sem_t m;
sem_init(&m,0,?);
sem_wait(&m);
section critique
sem_post(&m);
```

6.3 Exemple 1 : Rendez-vous

Le sémaphore permet aussi à un programme d'attendre qu'un autre programme se termine :

```
sem_t s;

void* child(void* arg){
   printf("childs\n);
   sem_post(&s);
   return NULL;
}

int main(int argc, char** argv){
   sem_init(&s,0,?);
   printf("parent begins\n");
   pthread_t c;
   pthread_create(&c,NULL,child,NULL);
   sem_wait(&s);
   printf("parent ends");
   return 0;
}
```