Chapitre 2 : Synchronisation de processus.

- Problème de l'accès concurrent à des ressources et sections critiques (Problème de l'exclusion mutuelle)
- Outils de synchronisation :

  - Evénements, Verrous
  - Sémaphores
  - Moniteurs
  - Régions critiques.
  - Expressions de chemins

1. INTRODUCTION :

Lorsque deux ou plusieurs processus doivent partager un objet, un mécanisme d'arbitrage est nécessaire pour qu'ils n'essaient pas de l'utiliser en même temps. L'objet particulier partagé n'a pas un grand impact sur le choix de tels mécanismes.

Le problème de l'évitement des conditions de course peut également être formulé de manière abstraite. Une partie du temps, un processus est occupé à faire des calculs internes et d'autres choses qui ne conduisent pas à des conditions de course. Cependant, il arrive qu'un processus accède à la mémoire partagée ou à des fichiers, ou qu'il fasse d'autres choses critiques qui peuvent conduire à des situations de concurrence. La partie du programme où l'on accède à la

mémoire partagée est appelée la **section critique (CS)**. Si nous pouvions faire en sorte que deux processus ne se trouvent jamais dans leur section critique en même temps, nous pourrions éviter les situations de concurrence.

Bien que cette exigence permette d'éviter les conditions de course, elle n'est pas suffisante pour que les processus parallèles coopèrent correctement et efficacement en utilisant des données partagées. Quatre conditions doivent être remplies pour obtenir une bonne solution :

- Deux processus ne peuvent pas se trouver simultanément à l'intérieur de leurs sections critiques.
- Aucune hypothèse ne peut être faite sur les vitesses ou le nombre de processeurs.
- Aucun processus fonctionnant en dehors de sa CS ne peut bloquer d'autres processus.
- Aucun processus ne devrait avoir à attendre éternellement pour entrer dans son CS.

✓ **Exemple** : Considérez les exemples suivants :

- Deux processus partageant une imprimante doivent l'utiliser à tour de rôle ; s'ils tentent de l'utiliser simultanément, la sortie des deux processus peut être mélangée dans un fouillis arbitraire qui ne sera probablement d'aucune utilité.
- Deux processus tentant de mettre à jour le même compte bancaire doivent se succéder ; si chaque processus lit le solde actuel à partir d'une base de données, le met à jour, puis le réécrit, l'une des mises à jour sera perdue.

Les deux exemples ci-dessus peuvent être résolus s'il existe un moyen pour chaque processus d'exclure l'autre de l'utilisation de l'objet partagé pendant les sections critiques du code. Ainsi, le problème général est décrit comme le problème de l'exclusion mutuelle. Les utilisateurs doivent faire face à un certain nombre de nouveaux problèmes sur les systèmes qui permettent de construire des programmes à partir de plusieurs processus simultanés. Certains de ces problèmes ont été largement étudiés et un certain nombre de solutions sont connues.

Bien que la plupart des systèmes d'exploitation ne prennent en charge que quelques mécanismes de base pour aider à résoudre les problèmes de programmation simultanée, de nombreux langages de programmation expérimentaux ont incorporé de tels mécanismes. Certaines de ces langues ont dépassé le stade expérimental et pourraient devenir largement utilisées ; parmi ceux-ci, le **shell** Unix et le langage de programmation **Ada** sont particulièrement importants.

La notion de **synchronisation** des processus est indispensable pour un système d'exploitation efficace et robuste. Sur la base de la synchronisation, les processus sont classés dans l'un des deux types suivants :

- **Processus indépendant** : l'exécution d'un processus n'affecte pas l'exécution des autres processus.
- **Processus coopératif** : L'exécution d'un processus affecte l'exécution d'autres processus.

Le problème de synchronisation des processus se pose également dans le cas du processus coopératif parce que les ressources sont partagées dans les processus coopératifs.

2. CONDITION DE CONCURRENCE

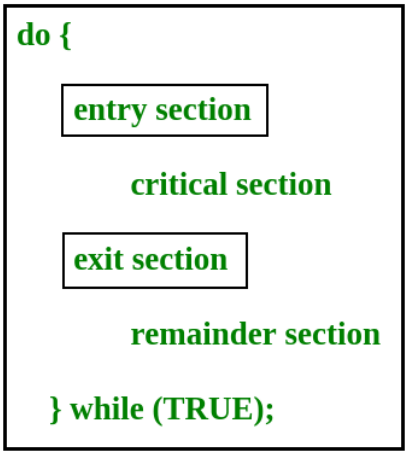
Lorsque plusieurs processus exécutent le même code ou accèdent à la même mémoire ou à toute variable partagée dans cette condition, il est possible que la sortie ou la valeur de la variable partagée soit erronée, de sorte que tous les processus faisant la course pour dire que ma sortie est correcte cette condition connue comme une condition de course. Plusieurs processus accèdent et traitent les manipulations sur les mêmes données simultanément, puis le résultat dépend de l'ordre particulier dans lequel l'accès a lieu.

Une condition de concurrence est une situation qui peut se produire à l'intérieur d'une section critique. Cela se produit lorsque le résultat de l'exécution de plusieurs threads dans la section critique diffère selon l'ordre dans lequel les threads s'exécutent.

Les conditions de concurrence dans les sections critiques peuvent être évitées si la section critique est traitée comme une instruction atomique. En outre, une synchronisation correcte des threads à l'aide de verrous ou de variables atomiques peut empêcher les conditions de concurrence.

3. PROBLEME DE SECTION CRITIQUE

La section critique est un segment de code accessible par un seul processus à la fois. La section critique contient des variables partagées qui doivent être synchronisées pour maintenir la cohérence des variables de données.



Dans la section d'entrée, le processus demande une entrée dans la **section critique.** Toute solution au problème de la section critique doit satisfaire trois exigences :

- **Exclusion mutuelle** : si un processus s'exécute dans sa section critique, aucun autre processus n'est autorisé à s'exécuter dans la section critique.
- **Progression** : si aucun processus ne s'exécute dans la section critique et que d'autres processus attendent en dehors de la section critique, seuls les processus qui ne s'exécutent pas dans leur section restante peuvent participer à la décision de celui qui entrera ensuite dans la section critique, et la sélection ne peut pas être reportée indéfiniment.
- **Attente limitée** : une limite doit exister sur le nombre de fois où d'autres processus sont autorisés à entrer dans leurs sections critiques après qu'un processus a fait une demande pour entrer dans sa section critique et avant que cette demande ne soit accordée.

4. SOLUTION DE PETERSON

La solution de Peterson est une solution logicielle classique au problème de la section critique. Dans la solution de Peterson, nous avons deux variables partagées :

- ***boolean flag[i]*** : Initialisé à FALSE, initialement aucun processus n'est intéressé à entrer dans la SC ;
- ***int turn*** **:** Le processus dont le tour est d'entrer dans la section critique.

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

} while (TRUE) ;
```

La solution de Peterson préserve les trois conditions :

- L'exclusion mutuelle est assurée car un seul processus peut accéder à la section critique à tout moment.
- La progression est également assurée, car un processus en dehors de la section critique n'empêche pas les autres processus d'entrer dans la section critique.
- L'attente limitée est préservée car chaque processus a une chance équitable.

Inconvénients de la solution de Peterson

- Cela implique une attente occupée
- Il est limité à deux processus.

5. L'ALGORITHME DE DEKKER

Cela a été réalisé pour la première fois par TJ Dekker et publié (par Dijkstra) en 1965. L'algorithme de Dekker utilise l'attente occupée et ne fonctionne que pour deux processus. L'idée de base est que les processus enregistrent leurs besoins à entrer dans une section critique (dans des variables booléennes) et qu'ils se relaient (à l'aide d'une variable appelée "tour") lorsque les deux ont besoin d'une entrée en même temps. La solution de Dekker est illustrée ci-dessus.

```
Var  process1, process2 :(interieur, exterieur);
        tour  : 1..2;
  process1:= exterieur;
  process2:= exterieur;
```

Processus P1

```
    ----------
process1:= interieur;
tour:= 2;
test: tantque process2 = interieur et tour = 2 faire
        aller à test
       Fintantque
 <section critique>
process1:= exterieur;
  ----------
```

Processus P2

```
    ----------
process2 := interieur;
tour := 1;
test: tantque process1 = interieur et tour = 1 faire
            allerà test
      Fintantque
<section critique>;
process2:=exterieur;
  ----------
```

La solution de **Dekkers** au problème d'exclusion mutuelle nécessite que chacun des processus en conflit ait un identifiant de processus unique appelé "moi" qui est transmis aux opérations d'attente et de signal.

Il suppose que si deux processus tentent d'écrire deux valeurs différentes dans le même emplacement mémoire en même temps, l'une ou l'autre valeur sera stockée, et non un mélange des deux. C'est ce qu'on appelle l'hypothèse de mise à jour atomique.

6. SEMAPHORES

Le sémaphore a été proposé par Dijkstra en 1965, une technique très importante pour gérer des processus concurrents en utilisant une simple valeur entière, connue sous le nom de sémaphore. Le sémaphore est simplement une variable entière partagée entre les threads. Cette variable est utilisée pour résoudre le problème de la section critique et pour réaliser la synchronisation des processus dans l'environnement multitraitement.

Un sémaphore est un mécanisme de signalisation, et un thread qui attend un sémaphore peut être signalé par un autre thread. Ceci est différent d'un *mutex* car le *mutex* ne peut être signalé que par le thread qui a appelé la fonction d'attente.

- Un sémaphore utilise deux opérations atomiques, attendre et signaler pour la synchronisation des processus.
- Un sémaphore est une variable entière, accessible uniquement via deux opérations :
  - *wait()*
  - *signal()*

Les sémaphores peuvent être utilisés pour résoudre le problème d'exclusion mutuelle au niveau de l'utilisateur ; par exemple, considérez les fragments de code illustrés en bas.

```
var shared: ...      { the shared variable needing mutual
    mutex: semaphore { a shared semaphore; initial count
      .
      .
      .

repeat { a typical process using shared }
      .
      .     { code outside the critical section;
      .       \dreferences to shared are not allowed }\u

      wait( mutex ) { start of critical section };
          .
          .     { code inside the critical section;
          .       \dreferences to shared are allowed }\u

      signal( mutex ) { end of critical section };
      .
      .     { more code outside the critical section }
      .

until ...
```

Ce code utilise un sémaphore pour contrôler l'accès à une variable partagée, mais le sémaphore pourrait tout aussi bien être utilisé pour contrôler l'accès à un fichier ou un périphérique partagé.

6.1. TYPE DE SEMAPHORES

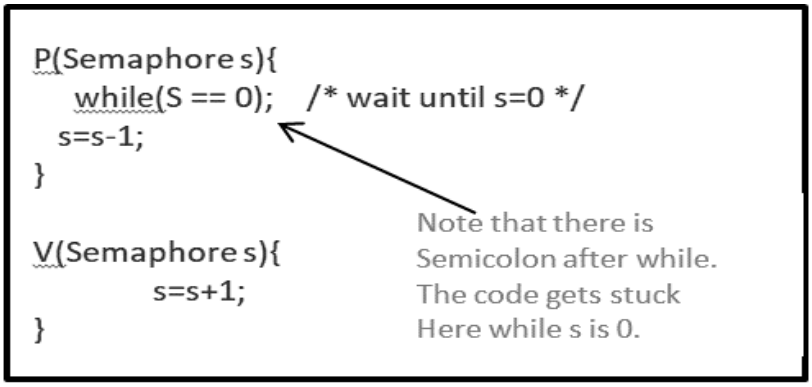
Il existe deux types de sémaphores : les sémaphores binaires et les sémaphores de comptage.

- ***Sémaphores binaires*** : ils ne peuvent être que 0 ou 1. Ils sont également connus sous le nom de verrous mutex, car les verrous peuvent fournir une exclusion mutuelle. Tous les processus peuvent partager le même sémaphore mutex initialisé à 1. Ensuite, un processus doit attendre que le verrou devienne 0. Ensuite, le processus peut rendre le sémaphore mutex 1 et démarrer sa section critique. Lorsqu'il termine sa section critique, il peut réinitialiser la valeur du sémaphore mutex à 0 et un autre processus peut entrer dans sa section critique.

- ***Compter les sémaphores*** : ils peuvent avoir n'importe quelle valeur et ne sont pas limités à un certain domaine. Ils peuvent être utilisés pour contrôler l'accès à une ressource dont le nombre d'accès simultanés est limité. Le sémaphore peut être initialisé au nombre d'instances de la ressource. Chaque fois qu'un processus veut utiliser cette ressource, il vérifie si le nombre d'instances restantes est supérieur à zéro, c'est-à-dire si le processus a une instance disponible. Ensuite, le processus peut entrer dans sa section critique, diminuant ainsi la valeur du sémaphore de comptage de 1. Une fois le processus terminé avec l'utilisation de l'instance de la ressource, il peut quitter la section critique, ajoutant ainsi 1 au nombre d'instances disponibles. de la ressource.

Tout d'abord, regardez deux opérations qui peuvent être utilisées pour accéder et modifier la valeur de la variable sémaphore.

Un sémaphore simple ou binaire est un sémaphore qui n'a que deux valeurs, zéro et un ; lorsqu'elle est utilisée pour résoudre le problème d'exclusion mutuelle, la valeur zéro indique qu'un processus à l'usage exclusif de la ressource associée et la valeur un indique que la ressource est libre. Alternativement, les états du sémaphore sont parfois nommés "réclamés" et "libres" ; l'opération d'attente réclame le sémaphore, l'opération de signal le libère. Les sémaphores binaires qui sont implémentés à l'aide de boucles d'attente occupées sont parfois appelés verrous tournants car un processus qui attend l'entrée d'une section critique passe son temps à tourner autour d'une boucle d'interrogation très serrée.

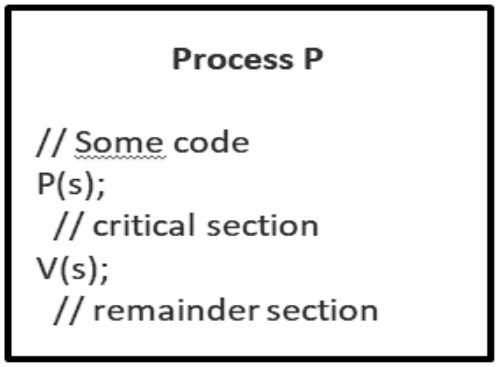


Quelques points concernant le fonctionnement *P* et *V* :

L'opération *P* est également appelée opération d'attente, de veille ou d'arrêt, et l'opération *V* est également appelée opération de signal, de réveil ou d'activation.

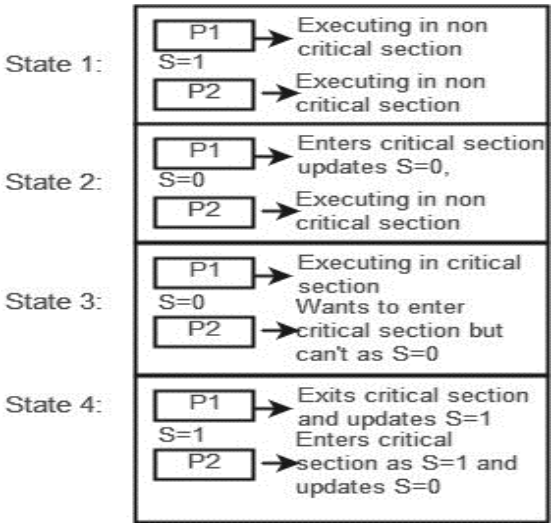
Les deux opérations sont atomiques et le(s) sémaphore(s) est toujours initialisé à un. Ici, atomique signifie que la variable sur laquelle la lecture, la modification et la mise à jour se produisent en même temps/moment sans préemption, c'est-à-dire qu'entre la lecture, la modification et la mise à jour, aucune autre opération n'est effectuée qui puisse modifier la variable.

Une section critique est entourée par les deux opérations pour mettre en œuvre la synchronisation des processus. Voir l'image ci-dessous. La section critique du processus P se situe entre les opérations P et V.



Voyons maintenant comment il met en œuvre l'exclusion mutuelle. Soit deux processus P1 et P2 et un sémaphore s est initialisé à 1. Maintenant, si P1 entre dans sa section critique, la valeur du sémaphore s devient 0. Maintenant, si P2 veut entrer dans sa section critique, il attendra jusqu'à ce que s > 0, cela ne peut se produire que lorsque P1 termine sa section critique et appelle l'opération V sur le sémaphore s.

De cette façon, l'exclusion mutuelle est réalisée. Regardez l'image ci-dessous pour plus de détails sur le sémaphore binaire.



6.2. IMPLEMENTATION DES SEMAPHORES BINAIRES

```
struct semaphore {
                enum value(0, 1);
                // q contains all Process Control Blocks (PCBs)
                // corresponding to processes got blocked
                // while performing down operation.
                Queue<process> q;
    } P(semaphore s)
    {
                if (s.value == 1) {
                        s.value = 0;
                }
                else {
                        // add the process to the waiting queue
                        q.push(P)
                        sleep();
                }
    }
    V(Semaphore s)
    {
                if (s.q is empty) {
                        s.value = 1;
                }
                else {
                        // select a process from waiting queue
                        Process p=q.pop();
                        wakeup(p);
    }
    }
```

La description ci-dessus concerne un sémaphore binaire qui ne peut prendre que deux valeurs 0 et 1 et assurer une exclusion mutuelle. Il existe un autre type de sémaphore appelé sémaphore de comptage qui peut prendre des valeurs supérieures à un.

Supposons maintenant qu'il existe une ressource dont le nombre d'instances est de 4. Maintenant, nous initialisons S = 4 et le reste est le même que pour le sémaphore binaire. Chaque fois que le processus veut cette ressource, il appelle P ou attend la fonction et quand c'est fait, il appelle V ou la fonction de signal. Si la valeur de S devient nulle, un processus doit attendre que S devienne positif. Par exemple, supposons qu'il y ait 4 processus P1, P2, P3, P4, et qu'ils appellent tous l'opération d'attente sur S (initialisé avec 4). Si un autre processus P5 veut la ressource, il doit attendre que l'un des quatre processus appelle la fonction signal et que la valeur du sémaphore devienne positive.

- **Limite** :

L'une des plus grandes limitations du sémaphore est l'inversion de priorité.

Interblocage, supposons qu'un processus essaie de réveiller un autre processus qui n'est pas en état de veille. Par conséquent, un interblocage peut bloquer indéfiniment. Le système d'exploitation doit garder une trace de tous les appels à attendre et à signaler le sémaphore.

Le principal problème avec les sémaphores est qu'ils nécessitent une attente occupée. Si un processus se trouve dans la section critique, les autres processus essayant d'entrer dans la section critique attendront jusqu'à ce que la section critique ne soit occupée par aucun processus.

- Chaque fois qu'un processus attend, il vérifie en permanence la valeur du sémaphore (regardez cette ligne pendant que (s==0); en fonctionnement P) et gaspille le cycle CPU.
- Il existe également un risque de "spinlock" car les processus continuent de tourner en attendant le verrouillage.

Pour éviter cela, une autre implémentation est fournie ci-dessous.

- Implémentation du sémaphore de comptage :

```
struct Semaphore {
                        int value;
                        // q contains all Process Control Blocks(PCBs)
                        // corresponding to processes got blocked
                        // while performing down operation.
                        Queue<process> q;

} P(Semaphore s)
{
                        s.value = s.value - 1;
                        if (s.value < 0) {
                                // add process to queue
                                // here p is a process which is currently executing
```

```
                        q.push(p);
                        block();
                }
                else
                        return;
}
V(Semaphore s)
{
                s.value = s.value + 1;
                if (s.value >= 0) {

                        // remove process p from queue
                        Process p=q.pop();
                        wakeup(p);
                }
                else
                        return;
}
```

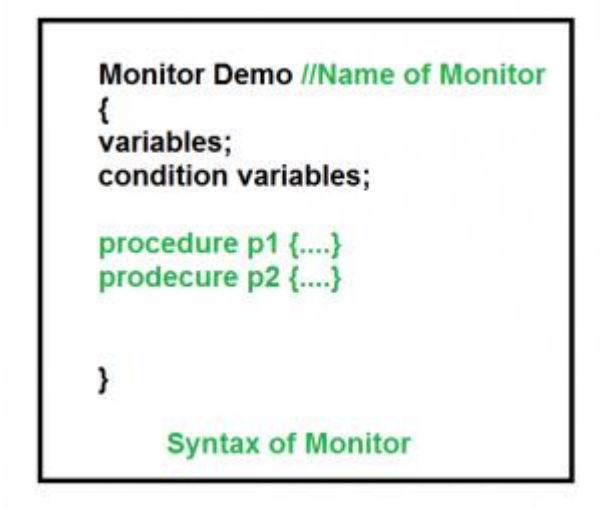
Dans cette implémentation, chaque fois que le processus attend, il est ajouté à une file d'attente de processus associés à ce sémaphore. Cela se fait via l'appel système block() sur ce processus. Lorsqu'un processus est terminé, il appelle la fonction signal et un processus de la file d'attente est repris. Il utilise l'appel système wakeup().

7. MONITEUR DANS LA SYNCHRONISATION DE PROCESSUS

Le moniteur est l'un des moyens de réaliser la synchronisation des processus. Le moniteur est pris en charge par des langages de programmation pour réaliser une exclusion mutuelle entre les processus. Par exemple, les méthodes Java Synchronized. Java fournit les constructions wait() et notify().

1. Il s'agit d'un ensemble de variables de condition et de procédures combinées dans un type spécial de module ou de package.
2. Les processus exécutés en dehors du moniteur ne peuvent pas accéder à la variable interne du moniteur mais peuvent appeler des procédures du moniteur.
3. Un seul processus à la fois peut exécuter du code à l'intérieur des moniteurs.

7.1. Syntaxe:



Deux opérations différentes sont effectuées sur les variables de condition du moniteur.

- ✓ Wait
- ✓ signal

Opération

x.wait() Le processus effectuant une opération d'attente sur n'importe quelle variable de condition est suspendu. Les processus suspendus sont placés dans la file d'attente des blocs de cette variable de condition.

Chaque variable de condition à sa file d'attente de blocs unique.

x.signal() : lorsqu'un processus effectue une opération de signal sur une variable de condition, l'un des processus bloqués a une chance.

```
If (x block queue empty)
  // Ignore signal
else
  // Resume a process from block queue.
```

7.2. Avantages de Monitor : Les moniteurs ont l'avantage de rendre la programmation parallèle plus facile et moins sujette aux erreurs que l'utilisation de techniques telles que le sémaphore.

7.3. Inconvénients de Monitor : Les moniteurs doivent être implémentés dans le cadre du langage de programmation. Le compilateur doit générer du code pour eux. Cela donne au compilateur la charge supplémentaire de savoir quelles fonctionnalités du système d'exploitation sont disponibles pour contrôler l'accès aux sections critiques dans les processus simultanés. Certains langages prenant en charge les moniteurs sont Java, C #, Visual Basic, Ada et Euclid simultané.

8. SOLUTION MATERIELLE « TESTANDSET »

TestAndSet est une solution matérielle au problème de synchronisation. Dans TestAndSet, nous avons une variable de verrouillage partagé qui peut prendre l'une des deux valeurs, 0 ou 1.

0	**Déverrouiller**
1	Verrouiller

Avant d'entrer dans la section critique, un processus s'enquiert du verrou. S'il est verrouillé, il continue d'attendre jusqu'à ce qu'il se libère et s'il n'est pas verrouillé, il prend le verrou et exécute la section critique.

Dans *TestAndSet*, l'exclusion mutuelle et la progression sont préservées, mais l'attente limitée ne peut pas être préservée.

```
TAS(C)
   begin
      < verrouiller l'accès à C >;
       lire C
      if  C = 0 then  begin
                        C := 1
                        co := co + 2   % co = compteur  ordinal %   (1)
                       end             % ou compteur d'instructions %
            else co := co + 1          %        (2)                 %
      endif
        < libérer l'accès à C >
   end
```

Par exemple, la programmation de l'exemple précédent peut être :

```
        Processus Debit                    Processus Credit
             .                                  .
             .                                  .
             .                                  .
Test: TAS(C);                      Test: TAS(C);
       Aller a Test;                      aller a Test;
      a1:  Rla := Compte  ;              b1:  Rlb := Compte  ;
      a2:  Rla := Rla + C ;              b1:  Rlb := Rlb - D ;
      a3:  Compte := Rla  ;              b3:  Compte := Rlb  ;
       C := 0;                            C := 0;
```

- **Limites**
  - **Attente active** (« busy waiting ») Verrou actif (« spin lock »)
  - **Inversion de priorités :** un processus prioritaire bloqué derrière un processus standard