Synchronisation de processus
Partie 2 : Les moniteurs

- Qu'est ce qu'un moniteur ?
- Variables de condition des moniteurs
- Exemple : Problème Producteur/consommateur
- Implémentation des moniteurs (au moyen de sémaphores)
- Moniteurs JAVA

Qu'est ce qu'un moniteur ?

- Moniteur = { variables globales + sections critiques d'un même problème}
- Pour assurer l'exclusion mutuelle, à tout moment un processus au plus est actif dans le moniteur.
- Si un processus est actif dans le moniteur et un autre demande à y accéder (appel une procédure du moniteur), ce dernier est mis en attente.
- C'est le compilateur qui se charge de cette tâche. Pour ce faire, il rajoute au moniteur le code nécessaire pour assurer l'exclusion mutuelle.
- Cette solution est plus simple que les sémaphores et les compteurs d'événements, puisque le programmeur n'a pas à se préoccuper de contrôler les accès aux sections critiques.
- La majorité des compilateurs utilisés actuellement ne supportent pas les moniteurs (à l'exception de JAVA, voir les exemples sur le site).

Qu'est ce qu'un moniteur ? (2)
Risque d'interblocage

- Le processus actif dans le moniteur a besoin d'une ressource détenue par un autre en attente du moniteur.
- Pour éviter des situations d'interblocage, il faut trouver un moyen permettant à un processus actif à l'intérieur d'un moniteur de se suspendre pour laisser place à un autre.

Exemple : Producteur/consommateur

- Le consommateur est actif dans le moniteur et le tampon est vide

=> il devrait se mettre en attente et laisser place au producteur.

- Le producteur est actif dans le moniteur et le tampon est plein

=> il devrait se mettre en attente et laisser place au consommateur.

Qu'est ce qu'un moniteur ? (2)
Risque d'interblocage

```
Moniteur ProducteurConsommateur
{    int compteur =0, ic=0, ip=0 ;

     void mettre (int objet)        // section critique pour le dépôt
     {       while (compteur==N);
             tampon[ip] = objet ;
             ip = (ip+1)%N ;
             compteur++ ;
      }

     void retirer (int* objet)      //section critique pour le retrait
     {       while (compteur ==0) ;
             objet = tampon[ic] ;
             ic = (ic+1)%N ;
             compteur -- ;
     }
}
```

Attente infinie dans le moniteur si le consommateur accède en premier alors que le tampon est vide.
Le producteur est en attente du moniteur.

Variables de condition des moniteurs

- Une variable de condition est une variable booléenne manipulée au moyen de deux opérations wait and signal.
- wait(x) :
  - suspendre l'exécution du processus (thread) appelant (le mettre en attente de x);
  - autoriser un autre processus en attente du moniteur à y entrer;
- signal(x) :
  - débloquer un processus en attente de la condition x.
- Le processus débloqué est soit :
  - mis dans la file d'attente du moniteur (signal and continue),
  - activé dans le moniteur (signal and wait) => Le processus appelant est dans ce cas mis en attente.

Exemple :
Problème Producteur/consommateur

- Les sections critiques du problème du producteur et du consommateur sont les opérations de dépôt et de retrait du tampon partagé.
- Un dépôt est possible uniquement si le tampon n'est pas plein. Pour bloquer le producteur tant que le buffer est plein, il suffit d'utiliser une variable de condition nplein et de précéder chaque opération de dépôt par l'action wait(nplein), si le tampon est plein.
- L'action signal(nplein) sera appelée suite à un retrait d'un buffer plein.
- Un retrait est possible uniquement si le tampon n'est pas vide. Pour bloquer le consommateur tant que le buffer est vide, il suffit d'utiliser une variable de condition nvide et de précéder chaque opération de retrait par l'action wait(nvide), si le tampon est vide.
- L'action signal(nvide) sera appelée par l'opération de dépôt dans le cas d'un dépôt dans un buffer vide.

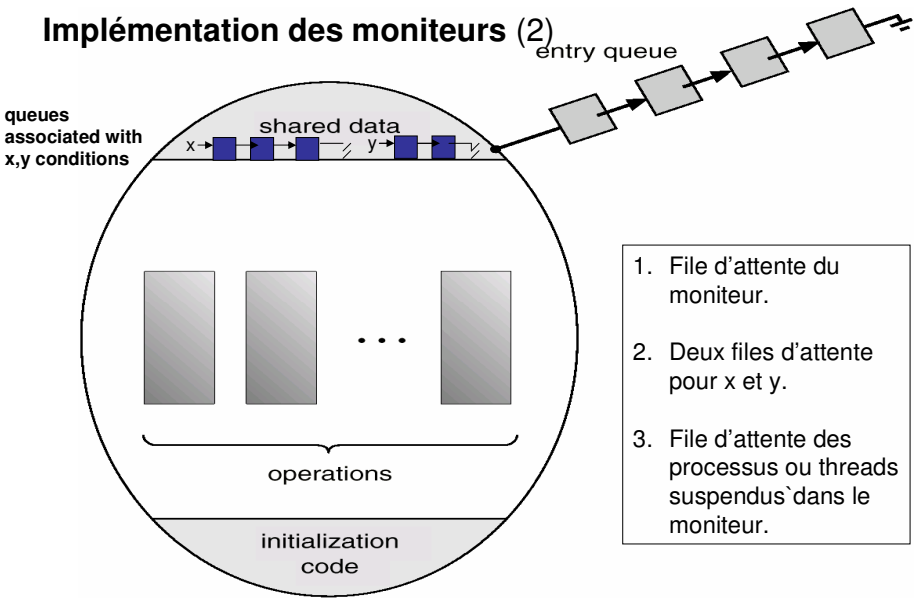
Exemple : Problème Producteur/consommateur (2)

```
Moniteur ProducteurConsommateur
{   boolc nplein, nvide ; //variable conditionnelle pour non plein et non vide
    int compteur =0, ic=0, ip=0 ;

    void mettre (int objet)       // section critique pour le dépôt
    {       //attendre jusqu'à ce que le tampon devienne non plein
            if (compteur==N) wait(nplein) ;
            tampon[ip] = objet ;
            ip = (ip+1)%N ;
            compteur++ ;
      // si le tampon était vide, envoyer un signal pour réveiller le consommateur.
            if (compteur==1) signal(nvide) ;
    }

    void retirer (int* objet)       //section critique pour le retrait
    {       if (compteur ==0) wait(nvide) ;
            objet = tampon[ic] ;
            ic = (ic+1)%N ;
            compteur -- ;
            // si le tampon était plein, envoyer un signal pour réveiller le producteur.
            if(compteur==N-1) signal(nplein) ;
    }
}
```


Implémentation des moniteurs

- Une file d'attente pour mémoriser les demandes d'accès au moniteur (file d'attente du moniteur).
- Une file d'attente pour chaque variable de condition (wait(x)).
- Éventuellement, une file d'attente des processus suspendus`dans le moniteur suite à l'opération signal(x) (dans le cas de signal and wait)
- Si cette dernière file est gérée, elle est plus prioritaire que la file d'attente du moniteur.

Implémentation des moniteurs (2)



1. File d'attente du moniteur.
2. Deux files d'attente pour x et y.
3. File d'attente des processus ou threads suspendus`dans le moniteur.

Implémentation des moniteurs (3) au moyen de sémaphores

- Un sémaphore binaire mutex, initialisé à 1, pour assurer l'accès en exclusion mutuelle au moniteur.
  File du mutex = File d'attente du moniteur
- Pour chaque variable de condition x du moniteur,
  - un sémaphore binaire x-sem initialisé à 0,
    File de x-sem = File d'attente de x
  - un compteur du nombre de processus en attente de x.
- Un sémaphore next, initialisé à 0, pour mémoriser tous les processus qui ont cédé le moniteur à un autre processus (signal and wait).
  File de next = File d'attente des processus suspendus
- Un compteur du nombre de processus suspendus à l'intérieur du moniteur next_count.

Implémentation des moniteurs (4) au moyen de sémaphores

Avant :

```
Function F (/* arguments.. */)
{
    /*corps de la fonction*/


}
```

Après :

```
Function F (/* arguments….*/)
{   P(mutex);
    /*corps de la fonction*/

    si (next_count >0)
    alors V(next);
    sinon V(mutex);
}
```


Implémentation des moniteurs (5) au moyen de sémaphores

Avant :

```
boolc x;

wait(x);
```

Après :

```
Semaphore x_sem=0;
int x_count = 0;

x_count = x_count + 1;
si(next_count>0)
alors          V(next);
sinon          V(mutex);

wait(x_sem);
x_count = x_count – 1;
```

Implémentation des moniteurs (6) au moyen de sémaphores

Avant :

signal(x); →

Après :

```
si (x_count > 0)
alors
{       V(x_sem);
        next_count = next_count + 1;
        P(next);
        next_count = next_count  – 1;
}
```


Moniteurs JAVA

- Un moniteur est associé à tout objet JAVA.
- Les méthodes de l'objet qui doivent accéder à l'objet en exclusion mutuelle sont déclarées de type « synchronized ».
- Une seule variable de condition «implicite » et les méthodes wait, notify (l'équivalent de signal) et notifyAll sont associées à tout objet JAVA.
- Les moniteurs JAVA sont de type signal and continue.
- Chaque objet JAVA a deux files d'attente :
  - Entry queue : file d'attente du moniteur
  - Wait queue : file d'attente de la variable de condition.

Interblocage

- Introduction
- Qu'est ce qu'un interblocage ?
- Conditions nécessaires pour l'interblocage
- Solutions au problème d'interblocage
  - La détection et la reprise
  - L'évitement des interblocages
  - La prévention des interblocages

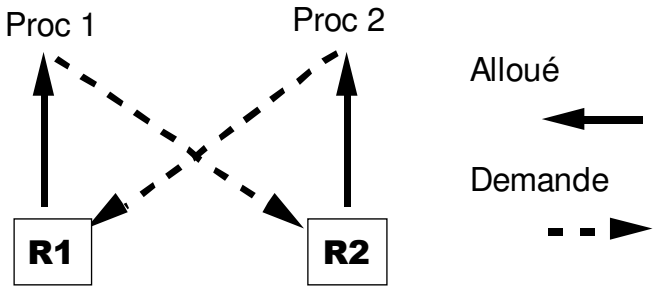
Introduction

- L'exécution d'un processus nécessite un ensemble de ressources (espace mémoire centrale, espace disque, fichier, périphériques, …) qui lui sont attribuées par le système d'exploitation.
- Des problèmes peuvent survenir, si des processus détiennent des ressources et en demandent d'autres qui sont déjà allouées.

Exemple,

- un processus Proc1 détient une ressource R1 et attend une autre ressource R2 qui est utilisée par un autre processus Proc2 ;
- le processus Proc2 détient la ressource R2 et attend la ressource R1.
- On a une situation d'**interblocage** (Proc1 attend Proc2 et Proc2 attend Proc1). Les deux processus vont attendre indéfiniment.

Qu'est ce qu'un interblocage ?



- Un ensemble de processus est en interblocage si chaque processus attend la libération d'une ressource allouée à un autre appartenant à l'ensemble.
- Comme tous les processus sont en attente, aucun ne pourra s'exécuter et donc libérer les ressources demandées par les autres. Ils attendront tous indéfiniment.

Conditions nécessaires pour interblocage (Coffman, Elphick et Shoshani)

- Exclusion mutuelle : une ressource est soit allouée à un seul processus, soit disponible ;
- Détention et attente : les processus qui détiennent des ressources peuvent en demander d'autres ;
- pas de réquisition : les ressources allouées à un processus sont libérées uniquement par le processus (ressources non préemptives);
- Attente circulaire: un ensemble de processus attendant chacun une ressource allouée à un autre.

Solutions au problème d'interblocage

- Les détecter et y remédier.
- Les éviter en allouant les ressources avec précaution. Si l'allocation d'une ressource peut conduire à un interblocage, elle est retardée jusqu'à ce qu'il n'y ait plus de risque.
- Les prévenir en empêchant l'apparition de l'une des quatre conditions nécessaires à leur existence.

Remarque :

- En général, ce problème est ignoré par les systèmes d'exploitation car le prix à payer pour les éviter ou les traiter est trop élevé pour des situations qui se produisent rarement.

La détection et la reprise

- Dans ce cas, le système ne cherche pas à empêcher les interblocages. Il tente de les détecter et d'y remédier.
- Pour détecter les interblocages, il construit dynamiquement le graphe d'allocation des ressources du système qui indique les attributions et les demandes de ressources.
- Le système vérifie s'il y a des interblocages :
  - A chaque modification du graphe suite à une demande d'une ressource (coûteuse en termes de temps processeur).
  - Périodiquement ou lorsque l'utilisation du processeur est inférieure à un certain seuil (la détection peut être tardive).

La détection et la reprise (2)
Graphe d'allocation des ressources

- Le graphe d'allocation des ressources est un graphe biparti composé de deux types de nœuds et d'un ensemble d'arcs :
  - Les processus qui sont représentés par des cercles ;
  - Les ressources qui sont représentées par des rectangles. Chaque rectangle contient autant de points qu'il y a d'exemplaires de la ressource représentée ;
  - Un arc orienté d'une ressource vers un processus signifie que la ressource est allouée au processus.
  - Un arc orienté d'un processus vers une ressource signifie que le processus est bloqué en attente de la ressource.
- Ce graphe indique pour chaque processus les ressources qu'il détient ainsi que celles qu'il demande.
- La détection est réalisée en réduisant le graphe.

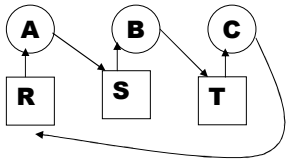
La détection et la reprise (3)
Graphe d'allocation des ressources

Exemple

- Soient trois processus A, B et C qui utilisent trois ressources R, S et T comme suit :

A	B	C
Demande R	Demande S	Demande T
Demande S	Demande T	Demande R
Libère R	Libère S	Libère T
Libère S	Libère T	Libère R

- Si les processus sont exécutés séquentiellement A suivi de B suivi C, il n'y pas d'interblocage.
- Supposons que l'exécution des processus est gérée par un ordonnanceur circulaire. On atteint une situation d'interblocage, si les instructions sont exécutées dans cet ordre :
  A demande R
  B demande S
  C demande T
  A demande S
  B demande T
  C demande R



Exercice 1

Considérons l'attribution des ressources suivante :

- A détient R et demande S ;
- B demande T ;
- C demande S ;
- D détient U et demande S et T ;
- E détient T et demande V ;
- F détient W et demande S ;
- G détient V et demande U.

Construire le graphe d'allocation des ressources. Y a-t-il un interblocage? Si oui, quels sont les processus concernés?

Exercice 2

- Considérons un système gérant quatre processus, P1 à P4, et trois types de ressources R1, R2 et R3 (3 R1, 2 R2 et 2 R3).
- L'attribution des ressources :
  - P1 détient une ressource de type R1 et demande une ressource de type R2 ;
  - P2 détient 2 ressources de type R2 et demande une ressource de type R1 et une ressource de type R3 ;
  - P3 détient 1 ressource de type R1 et demande une ressource de type R2 ;
  - P4 détient 2 ressources de type R3 et demande une ressource de type R1 ;

Construire le graphe d'allocation des ressources. Y a-t-il un interblocage ? Si oui, quels sont les processus concernés?

La reprise des interblocages

- Lorsque le système détecte un interblocage, il doit le supprimer, ce qui se traduit généralement par la réalisation de l'une des opérations suivantes :
  - Retirer temporairement une ressource à un processus pour l'attribuer à un autre.
  - Restaurer un état antérieur (retour arrière) et éviter de retomber dans la même situation.
  - Supprimer un ou plusieurs processus.
- La reprise n'est pas évidente.

L'évitement des interblocages

- Dans ce cas, lorsqu'un processus demande une ressource, le système doit déterminer si l'attribution de la ressource est sûre (mènent vers un état sûr).
- Si c'est le cas, il lui attribue la ressource.
- Sinon, la ressource n'est pas accordée.
- Un état est sûr si tous les processus peuvent terminer leur exécution (il existe un ordre d'allocation de ressources qui permet à tous les processus de se terminer).
- Il faut connaître à l'avance les besoins en ressources de chaque processus (ce qui est en général impossible).

Comment déterminer si un état est sûr ou non ? L'algorithme du banquier

- L'état est caractérisé par quatre tableaux.

$$E = \begin{pmatrix} R1 & R2 & R3 & R4 \\ 4 & 2 & 3 & 1 \end{pmatrix} \quad A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

$$Alloc = \begin{matrix} P1 \\ P2 \\ P3 \end{matrix} \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

$$Req = \begin{matrix} P1 \\ P2 \\ P3 \end{matrix} \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

1.trouver un processus Pi non marqué dont la rangée Pi de Req est inférieure à A ;

2.Si un tel processus n'existe pas alors l'état est non sûr. L'algorithme se termine.

3.Sinon, ajouter la rangée Pi de Alloc à A, marquer le processus ;

4.Si tous les processus sont marqués alors l'état est sûr et l'algorithme se termine, sinon aller à l'étape 1.

- Alloc : ressources attribuées ;
- Req : ressources nécessaires non encore obtenues ;
- A : ressources disponibles ;
- E : ressources du système

Exercice 3

- L'état courant est-il sûr ?
- Peut-on accorder 2 ressources R1 à P3 ?
- Peut-on accorder 1 ressource R1 à P2 ?

La prévention des interblocages

- Pour prévenir les interblocages, on doit faire de telle sorte que l'une des quatre conditions nécessaires à leur existence ne soit jamais satisfaite.
- Pas d'exclusion mutuelle : impossible car certaines ressources sont à usage exclusif.
- Pas de « détention et attente » : Il faudrait que toutes les ressources nécessaires à un processus soient demandées et allouées à la fois. Le processus ne doit pas détenir des ressources et en demander d'autres.
  - Il est difficile de prévoir les besoins du processus
  - Problème de famine.
- Préemption : n'est pas raisonnablement traitable pour la plupart des ressources sans dégrader profondément le fonctionnement du système. On peut cependant l'envisager pour certaines ressources dont le contexte peut être sauvegardé et restauré.

La prévention des interblocages (2)

- Pas d'attente circulaire, si on parvient à :
  - Établir un ordre total entre les ressources et
  - Imposer, à chaque processus, la règle de demande de ressources suivante :
    Un processus peut demander une ressource Rj seulement
    si toutes les ressources qu'il détient sont inférieures à Rj

Exemple : problème des philosophes

- Les fourchettes sont ordonnées f0 < f1 < f2 < f3 < f4.
- Les philosophes doivent prendre les fourchettes en ordre croissant :
  - Le philosophe 0 doit demander la fourchette f0 puis f4
  - Le philosophe 1 doit demander la fourchette f0 puis f1
  - Le philosophe 2 doit demander la fourchette f1 puis f2
  - Le philosophe 3 doit demander la fourchette f2 puis f3
  - Le philosophe 4 doit demander la fourchette f3 puis f4.

La prévention des interblocages (3)
Exemple : problème des philosophes

```
Semaphore Fourch[5] ={1,1,1,1,1}
```

```
Philosophe0()
{   penser();
    P(Fourch[0]);
    P(Fourch[4]);
    manger();
    V(Fourch[0]);
    V(Fourch[4]);
}
```

```
Philosophe (num in [1,4])
{           penser();
            P(Fourch[num-1]);
            P(Fourch[num]);
            manger();
            V(Fourch[num-1]);
            V(Fourch[num]);
}
```


Problème de famine

- Le problème de famine est un autre problème inhérent à la gestion de ressources.
- L'allocation de ressources est indéfiniment retardée.

Pour éviter ce problème :

- Mémoriser les demandes dans une file pour les traiter selon la discipline FIFO.
- Limiter le temps d'allocation de chaque ressource.
- Augmenter progressivement la priorité d'un processus avec le temps d'attente (si la priorité est un critère d'ordonnancement).

Exercice 4

- On dispose d'un mécanisme d'enregistrement à un ensemble de cours, tel que :
  - tout étudiant ne peut être inscrit qu'à au plus trois cours, et
  - chaque cours a un nombre limité de places.
- Un étudiant inscrit déjà à trois cours peut s'il le souhaite en abandonner un, pour en choisir un autre dans la limite des places disponibles.
- Si cet échange n'est pas possible, l'étudiant ne doit pas perdre les cours auxquels il est déjà inscrit.
- Le bureau des affaires académiques souhaite donc mettre en place un système de permutation de cours, permettant à un étudiant de changer de cours. Il vous sollicite pour vérifier si l'implémentation que vous avez proposée il y a un an (avant se suivre le cours INF3600) est correcte :

Exercice 4 (suite)

```
void EchangeCours (Putilisateur utilisateur, PCours cours1, cours2) {
    cours1->verrouille ();  // verrouille l'accès à l'objet cours1
    cours1->desinscrit (utilisateur);
    if (cours2->estPlein == false) {
            cours2->verrouille ();   // verrouille l'accès à l'objet cours2
            cours2->inscrit (utilisateur);
            cours2->deverrouille ();  //déverrouille l'accès à l'objet cours2
    }
    cours1->deverrouille ();  //déverrouille l'accès à l'objet cours1
}
```

- **Vérifiez si l'implémentation est correcte** : Si elle est correcte, **expliquez pourquoi**, en montrant comment est géré le cas où deux étudiants (ou plus) veulent accéder en même temps au système. Si elle est incorrecte**, listez et expliquez les problèmes**, et **proposez** une solution qui fonctionne.