Les paradigmes de la concurrence

PROBLEMES

- Contrôler les accès à des ressources partagées
- Déterminer l'ordre correct des lectures/écritures lors de l'exécution de plusieurs activités.

Les sémaphores permettent de programmer ces besoins mais leurs utilisations peuvent être délicates.

Solution :
Identifier les paradigmes pouvant servir de schéma de programmation

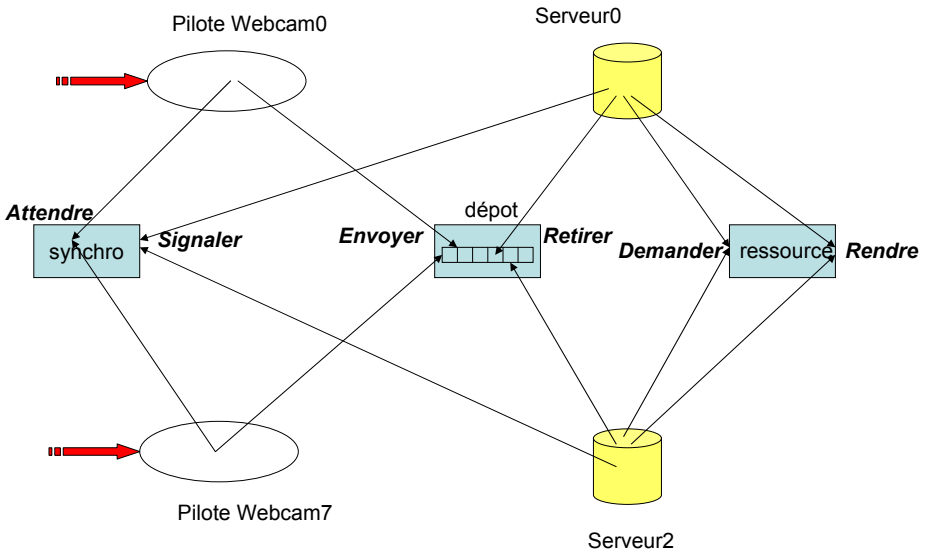
- L'exclusion mutuelle
  Réalisation d'une séquence d'instructions de façon atomique
- La cohorte
  N serveurs au plus coopèrent pour servir des clients
- Les producteurs/consommateurs
  Un tampon de N cases est écrit par une ou plusieurs tâches et lus par une plusieurs tâches

Paradigmes (suite)

- Les lecteurs/rédacteurs
  Une base de données est lue par un ensemble de lecteurs et écrite, en concurrence, par un et seul rédacteur.
- Le passage de témoin
  Envoi d'une information d'une tâche à une autre qui permet d'attendre pour l'une et de signaler pour l'autre que l'information soit disponible : notion de rendez-vous.
- Le repas des philosophes
  Accès compétitif à un ensemble de ressources (périphériques d'E/S) par un ensemble de tâches en évitant l'interblocage et la famine.

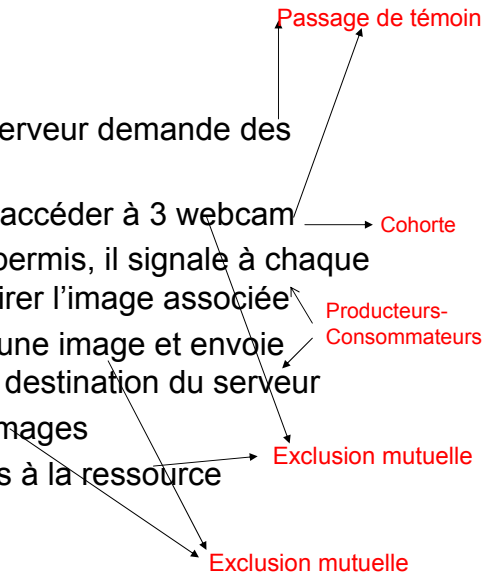
Etude de cas 1 : surveillance par Webcam

- Les activités :
  - 8 Pilotes qui gèrent 8 webcam
  - 3 Serveurs qui récupèrent les images : chaque serveur gère 3 webcam.
- La communication :
  - Chaque pilote dépose une image dans un dépôt commun
  - Chaque serveur demande à être autorisé à retirer les 3 images du dépôt
  - Le serveur autorisé retire les images.



- Les pilotes sont des clients
- Les serveurs sont … des serveurs

- La synchronisation :
  - Le pilote attend qu'un serveur demande des images
  - Un serveur demande d'accéder à 3 webcam
  - Lorsque l'accès lui est permis, il signale à chaque pilote qu'il est prêt à retirer l'image associée
  - Chaque pilote acquiert une image et envoie l'image dans le dépôt à destination du serveur
  - Le serveur retire les 3 images
  - Le serveur libère l'accès à la ressource



Algorithme

Un Client

```
Begin
 loop
   synchro.Attendre (X) ; // attente d'un signal envoyé à X par un serveur
   Lecture_de_la_Camera (Z) ; // acquisition de l'image bimap
   ….
   Préparation de la donnée avant rangement
    ….
   depot.Envoyer (X,Y) // envoi vers les serveurs de l'image Y prise par la Webcam X
 End loop ;
 End Un_Client ;
```

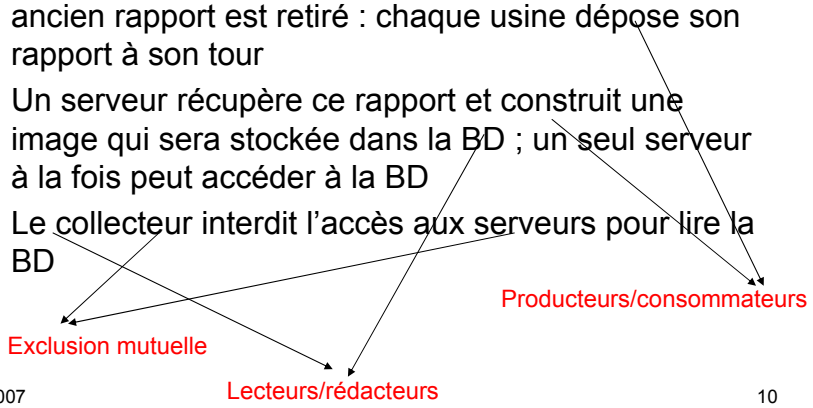
Un Serveur

```
Begin
 loop
   negociation_avec_le_client (A,B,C) ; // le client donne 3 noms de Webcam
   ressource.Demander(A) ; ressource.Demander(B) ; ressource.Demander(C) ; // réservation
                                                                          des Webcams
  synchro.Signaler(A) ; synchro.Signaler(B) ; synchro.Signaler(C) ;  // réveil des pilotes
  depot.Retirer(A,Y) ; // retrait de l'image de A et copie dans Y
  depot.Retirer(B,Z) ; // retrait de l'image de B et copie dans Z
  depot.Retirer(C,T) ; // retrait de l'image de C et copie dans T
  ressource.Rendre(A) ; ressource.Rendre(B) ; ressource.Rendre(C) ; // libération des Webcams
End loop ;
End Un_Serveur ;
```

Etude de cas N°2 : surveillance d'usines

- Les activités
  - 5 usines produisent un rapport d'activités
  - 3 serveurs récupèrent ces rapports
  - 1 collecteur les affiche
- La communication
  - Un rapport est transféré de chaque usine dans un serveur qui le transforme en une image stockée dans une base d'images
  - Périodiquement le collecteur affiche le contenu de la base d'images

- Un proxy associé à une usine produit des données
- Un serveur retire une donnée et demande l'accès au collecteur
- Le collecteur demande le droit d'accéder à la base
  - La synchronisation
    - Le retrait est possible que si l'envoi a été fait : le plus ancien rapport est retiré : chaque usine dépose son rapport à son tour
    - Un serveur récupère ce rapport et construit une image qui sera stockée dans la BD ; un seul serveur à la fois peut accéder à la BD
    - Le collecteur interdit l'accès aux serveurs pour lire la BD



Rappel : Producteur/Consommateur

- Producteur
  - produit
    - Avant rangement
    - Range
    - Après rangement
      - Signale produit
  - fin
- Consommateur
    - Avant prise
    - Prend
    - Après prise
      - consomme
  - fin

Soit n le nombre d'éléments du buffer

contrôle si :
buffer plein et rangement
buffer vide et consommation

Programmation : un producteur un consommateur

```
# include ''producteur_conso.h ''
# define BUFFER_SIZE 10          /* nb d'emplacements ds le tampon */
typedef int semaphore ;
semaphore vide ;
semaphore plein ;
item buffer [BUFFER_SIZE];
E0(vide, BUFFER_SIZE );          /* Nb ressources disponibles */
E0(plein, 0) ;                   /* envoi d'un signal */
  void producteur()
  {
    item objet_produit ;
   int in = 0 ;
    while (TRUE)  {
              produire_objet(&objet_produit) ;
              P(&vide) ;
              buffer[in] = objet_produit ;
              in = (in +1)%n ;
              V(&plein) ;
    }
  }
```

```
      void consommateur()
      {
        item objet_consomme ;
        int out = 0 ;
        while (TRUE)  {
                  P(&plein) ;
                   objet_consomme = buffer[out] ;
                   out = (out +1)%n ;
                  V(&vide) ;
                  consommer_objet(objet) ;
        }
      }
```

Le modèle Lecteurs-Rédacteurs

- Un ensemble de données est partagé par un ensemble de flots concurrents
  - Lecteurs – accès seulement en lecture ; pas de mises à jour
  - Rédacteurs – peuvent à la fois, lire et écrire
- Problème
  - Autoriser plusieurs lecteurs à lire les données au même moment.
  - Un seul rédacteur peut accéder aux données partagées à un instant donné.
  - Si un rédacteur écrit, aucun lecteur ne peut accéder aux données.
- Les données partagées
  - Un ensemble de données
  - Semaphore mutex.
  - Semaphore wrt.
  - Entier readcount initialisé à 0.

Lecteurs/rédacteurs

Read() {	Write() {
avant lire	creer données
lire	avant écrire
apres lire	écrire
utiliser données	apres écrire
{	{

Algorithme

Soit readcount le nb de lecteurs ; le lecteur a priorité sur le rédacteur

- Avant lire
  - Obtenir l'accès exclusif à readcount
  - readcount++
  - si readcount ==1 le 1er lecteur bloque la BD ; priorité aux lecteurs
  - Libérer l'accès exclusif à readcount
- Après lire
  - Obtenir l'accès exclusif à readcount
  - readcount--
  - si readcount ==0 signaler écriture // plus de lecteurs
  - Libérer l'accès exclusif à readcount
- Avant ecrire
  - Obtenir l'accès exclusif aux données
- Après ecrire
  - Libérer l'accès exclusif aux données

Programmation

- **La structure d'un flot lecteur**

```
while (true) {
        P (mutex) ;
            readcount ++ ;
            if (readcount == 1)  P (wrt) ;
        V (mutex)

                // la lecture est effectuée

        P (mutex) ;
            readcount  - - ;
            if (readcount  == 0)  V (wrt) ;
        V (mutex) ;
          utiliser_donnees();
}
```


Programmation (suite)

- **La structure d'un flot rédacteur**

```
while (true) {
            creer_donnees() ;
        P (wrt) ;

            //    l'écriture est effectuée

        V (wrt) ;
}
```

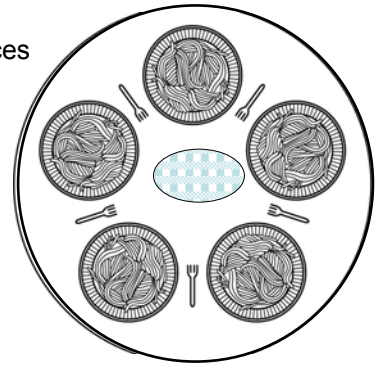
Problème ?

Que se passe t-il si un rédacteur attend pour écrire et que des lecteurs lisent en continuité ?

Les rédacteurs sont sujet à famine!

Le modèle du repas des philosophes

•Partage de plusieurs classes de ressources entre processus asynchrones
•Tout processus libère ses ressources au bout d'un temps fini
•Interblocage possible
•Famine possible



Données partagées :
Un plat de riz (ensemble de données)
Semaphores baguettes [N] d'exclusion mutuelle

Le repas des philosophes (1)

- n philosophes autour d'une table et n baguettes
- Les philosophes mangent (riz) puis pensent
- Manger nécessite 2 baguettes
- Chacun prend une baguette à un instant donné (d'abord à droite, puis à gauche puis il mange)

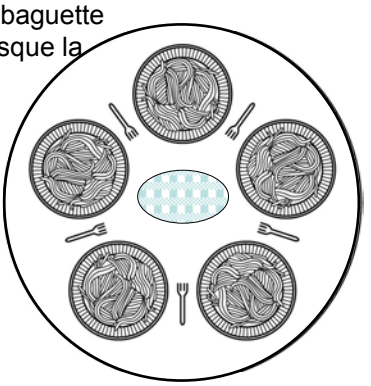
```
binarySemaphore baguettes[ ] = new binarySemaphore [ ] ;
For (int j = 0 ; j < n ; j++) baguettej[j] = new binarySemaphore (1) ;

Philosophe (int i /* 0..n-1 */ {
        while (true] {
        /*pense*/
         baguettes[i].P() ;          // prendre la baguette de gauche
         baguettes[(i+1)%n].P() ;    // prendre la baguette de droite
        /*manger*/
        baguettes[i].V() ;          // relâcher la baguette de gauche
         baguettes[(i+1)%n].V()     // relâcher la baguette de droite
        }
}
```

Problèmes ?

Le repas des philosophes (2)

Tous les philosophes peuvent prendre leur baguette gauche au même moment et se bloquer lorsque la baguette droite n'est pas accessible.



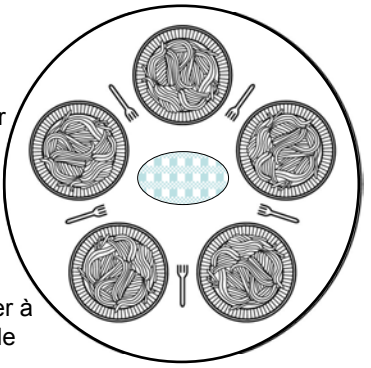
Première idée :
• prendre la baguette gauche ;
• si la droite n'est pas disponible alors rendre la gauche et attendre.

Il demeure un problème !!!!

Qu'arrive-t-il quand tous les philosophes font la même chose en même temps ?
- famine (plus personne ne progresse)

Le repas des philosophes (3)

Solution :
Utiliser un sémaphore d'exclusion mutuelle avant de prendre les deux baguettes et relâcher le sémaphore après avoir déposé les deux baguettes sur la table.



Est-ce une bonne solution?

Oui, mais seulement un philosophe peut manger à un instant donné, bien qu'il y ait suffisamment de baguettes pour deux.

Solution : autoriser seulement n-1 philosophes à agir simultanément

Programmation

```
binarySemaphore baguettes[ ] = new binarySemaphore [ ] ;
For (int j = 0 ; j < n ; j++) baguette[j] = new binarySemaphore (1) ;
binarySemaphore chaise = new binarySemaphore(n-1);

Philosophe (int i /* 0..n-1 */ {
        while (true] {
        /*pense*/
        chaise.P() ;                    // autoriser n-1 philosophes à table
         baguettes[i].P() ;             // prendre la baguette de gauche
         baguettes[(i+1)%n].P()         // prendre la baguette de droite
        /*manger*/
        baguettes[i].V() ;              // relâcher la baguette de gauche
         baguettes[(i+1)%n].V()         // relâcher la baguette de droite
        chaise V() ;                    // libérer un éventuel philosophe en attente
}
```


Accès compétitifs à des ressources partagées

- **Que faut-il retenir ?**
- D'abord, lorsque plusieurs tâches (flots) accèdent à des ressources partagées, il y a possibilité d'interblocage.
- Ensuite, attention à la famine : la seule solution est d'assurer que chaque flot ne puisse être bloqué de façon non maîtrisée ; une solution est de réaliser une file globale des flots bloqués et de définir un ordre de déblocage. Attention à l'impact sur les performances.
- Puis, se soucier de l'équité en faisant en sorte que chaque flots possède le même nombre de ressources.

Synthèse

- Tout problème de synchronisation peut être résolu par le concept de sémaphore.
- Les opérations sur sémaphores P et V sont utilisés pour d'une part signifier une exclusion mutuelle et d'autre part synchroniser des tâches.
- La valeur d'initialisation permet de différentier les différentes programmation des paradigmes.
- Rappel des paradigmes :
  - Exclusion mutuelle
  - Cohorte
  - Passage de témoin
  - Producteurs-consommateurs
  - Lecteurs-rédacteurs
  - Repas des philosophes

Propriétés globales d'une application concurrente

- Sûreté : garantir que rien de mauvais ne puisse se produire (exclusion mutuelle, pas d'interblocage)
- Vivacité : garantir que quelque chose de bon se produira.
  - Éviter la famine

  Ex: Prod/Cons : si service en pile ou par priorités

  Lect/réd. : si priorité aux lecteurs

  philos : si allocation globale des baguettes
  - Respecter les priorités imposées par l'appli.

- Exclusion mutuelle :

  Sémaphore S initialisé à 1.

  Le premier appel à P(S) autorise l'entrée, les autres appels sont bloqués tant que le premier n'a pas exécuté V(S).
- Cohorte : N serveurs, au plus, coopèrent

  Sémaphore S initialisé à N.

  N tâches peuvent passer P(S), en séquence, la N+1 ième étant bloquée. Lorsqu'une tâche quitte la cohorte, une tâche bloquée peut alors y entrer.
- Passage de témoin : Définition d'une autorisation qui n'est pas disponible initialement

  Sémaphore S initialisé à 0.

  La tâche qui attend un signal fait P(S) et l'émetteur fait V(S) qui signifie l'envoie d'une autorisation.

  Toutes les tâches sont bloquées tant qu'une autre n'a pas émis un V(S); le nombre de V(S) émis avant l'exécution d'un P(S) modélise une file de signaux en attente.

- Producteurs-Consommateurs : Contrôle de flux

  Le producteur produit des données pour un consommateur. Le consommateur produit des cases vides pour le producteur.

  Deux sémaphores :

  Nvide, initialisé à N, contrôle l'allocation des N cases d'un tampon.

  Nplein, initialisé à 0, modélise l'envoie d'un signal pour signifier qu'une donnée est disponible dans le tampon.

  Le producteur réserve une case du tampon P(Nvide)

  Le producteur envoie un signal au consommateur signifiant qu'une donnée est disponible dans une case : V(Nplein)

  Le consommateur attend le signal qu'une case peut être lue : P(Nplein)

  Le consommateur informe le producteur qu'une case est libre pour ranger une nouvelle donnée : V(Nvide)

  Conséquences :

  – Une donnée ne peut être concurremment produite par un producteur et lue par un récepteur.

  – Le producteur et le consommateur ne peuvent être bloqués simultanément si les accès aux données sont faits dès que les conditions l'autorisent.

- Producteurs-Consommateurs : contrôle de cohérence
  - Un producteur et un consommateur

    Exclusion mutuelle pendant l'accès à chaque case du tampon réalisé par le contrôle de flux ; les index sur le tampon sont des variables locales et l'accès aux données est exclusif. Pas de synchronisation complémentaire.
  - Plusieurs producteurs et un consommateur

    L'index sur le tampon en écriture est partagé donc un **sémaphore d'exclusion mutuelle** pour protéger cet index dans les producteurs ; toutes les données écrites doivent l'être l'une après l'autre, sans trou.
  - Plusieurs producteurs et plusieurs consommateurs

    L'index sur le tampon en lecture est partagé, donc **un sémaphore d'exclusion mutuelle** est nécessaire dans les consommateurs.

- Lecteurs-Rédacteurs
  - N lecteurs concurrents et un seul rédacteur ayant un accès exclusif
    - Priorité aux lecteurs : un lecteur en attente peut être activé même si un rédacteur est éligible
    - Priorité aux rédacteurs : un rédacteur est activé dès que les conditions d'activation l'autorisent.
    - Egalité : il n'y a pas de préséance entre lecteurs et rédacteurs.

- Priorité aux lecteurs

  Comptage du nombre de lecteurs ;
  - **exclusion mutuelle** sur le comptage des lecteurs.
  - **exclusion mutuelle** entre les rédacteurs et entre un rédacteur et un groupe de lecteurs.
    - Un compteur NL, nombre de lecteurs protégé par un Mutex_L sur les lecteurs,
    - Mutex_A entre un rédacteur et un groupe de lecteurs.
  - exclusion mutuelle entre le 1er lecteur et un rédacteur.

  Mutex_L permet de ne bloquer que les lecteurs par P(Mutex_L) et n'empêche pas un rédacteur de réveiller un lecteur par un V(Mutex_A)
  P(Mutex_L) bloque les lecteurs seulement pour incrémenter NL
  V(Mutex_A) réveille un éventuel rédacteur en attente
  Le dernier lecteur réveille un éventuel rédacteur par un V(Mutex_A).

  Conséquence :
  Solution non équitable qui peut conduire à une famine pour les rédacteurs.

- Priorité égale entre lecteurs et rédacteurs

  Introduire un nouveau sémaphore **d'exclusion mutuelle** pour autoriser des rédacteurs à s'intercaler entre des lecteurs.

- Repas des philosophes
  - Introduction d'une cohorte de N-1 philosophes pour faire en sorte qu'au moins un philosophe qui a pris une baguette à gauche puisse prendre la baguette de droite.
    - Pas d'interblocage
    - Equitable : pas de famine

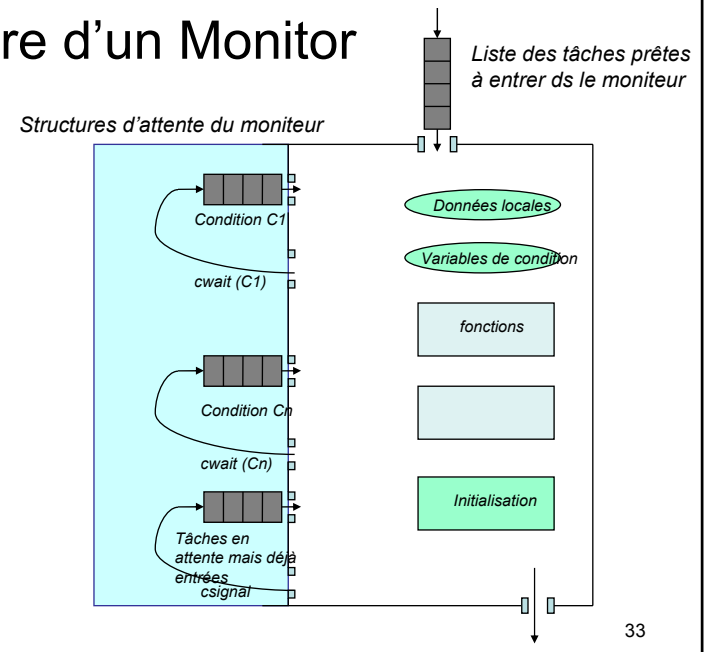
Remarques

- Les sémaphores et les variables partagées sont des variables globales : ils doivent être visibles de tout le programme ; peu approprié en programmation objet.
- Les opérations sur les sémaphores P() et V() expriment à la fois l'exclusion mutuelle et la synchronisation ce qui nécessite d'examiner le code pour voir à quoi un sémaphore est utilisé.
- Les moniteurs ont été proposées pour résoudre ces problèmes (Hoare, Brinch Hansen 1971) ; ils supportent l'encapsulation de données et les données cachées.

Les moniteurs

- Les moniteurs sont des constructions d'un langage de programmation qui assurent la protection des données partagées.
- Ils simplifient la mise en place de sections critiques
- Définis par :
  - Des données internes (variables d'état)
  - Des primitives d'accès aux moniteurs (points d'entrée)
  - Des primitives internes seulement accessibles de l'intérieur du moniteur
  - Une ou plusieurs files d'attente

Structure d'un Monitor



Sémantique

- Seule une tâche (ou flot) peut être active à un instant donné à l'intérieur du moniteur
- L'accès à un moniteur sera bloquant tant qu'il y aura une tâche active à l'intérieur du moniteur
  - L'accès à un moniteur construit donc implicitement une exclusion mutuelle aux ressources du moniteur
- Lorsqu'une tâche active au sein d'un moniteur est bloquée sur l'attente de ressources, le moniteur est libéré et la tâche bloquée est mise en attente
- Une tâche en attente doit être réveillée lorsque les variables internes du moniteur sont modifiées

Primitives wait et signal

- wait : met en attente l'appelant et libère l'accès au moniteur
- signal : réveille une des tâches en attente à l'intérieur du moniteur ; cette tâche a préalablement exécutée un wait.
- Implémentées de différentes façons selon les langages
  - Méthodes « wait/notify/notifyAll » en Java et méthodes synchronized
  - Primitives pthread_cond_wait/pthread_cond_signal en Posix et variables conditionnelles
- La sémantique des réveils peut varier :
  - Qui réveille t-on ? (le plus ancien, le plus prioritaire, au hasard,…)
  - Quand réveille t-on ? (dès la sortie du moniteur, au prochain ordonnancement , …)


```
/* program producteurconsommateur */
monitor boundedbuffer ;
int  buffer [N] ;                       /* tableau pour N entiers */
int nextin, nextout ;                   /* pointeurs sur le buffer */
int count ;                             /* nombre d'entiers possibles du buffer */
cond notfull, notempty ;                /* variables de condition pour synchronisation */

void append (int x) {
        if (count == N)
        cwait (notfull) ;               /* buffer est plein ; on attend */
        buffer[nextin] = x ;            /* le buffer est rempli */
        nextin = (nextin+1)%N ;         /* le pointeur est incrémenté modulo N */
        count ++ ;
        csignal (notempty) ;            /* réveille une tâche qui serait en attente
                                        sur buffer vide */
}
void take (int x) {
        if (count == 0)
        cwait (notempty) ;              /* le buffer est vide ; on attend */
        x = buffer [nextout] ;          /* le prochain caractère est lu */
        nextout = (nextout+1)%N ;       /* le pointeur est incrémenté modulo N */
        count - - ;
        csignal (notfull) ;             /* réveille une tâche qui serait en attente sur
                                        buffer plein */
{
    nextin = nextout = count = 0 ;
}
```

```
void producteur() {
int x ;
        while (true)
        {
        produce (x) ;
        append(x) ;
        }
}

void consommateur () {
int x ;
        while (true)
        {
        take(x) ;
        consume(x);
        }
}
void main ()
{
        parbegin(producer, consumer) ;
}
```


Moniteur en Java

java.util.concurrent

- Chaque objet en Java est associé à un seul verrou ; lorsque la méthode est déclarée *synchronised,* l'invocation de la méthode revient à accéder au verrou.

```
public class SimpleClass {
….
   public synchronized void safeMethod() {
     /* implémentation de safeMethod() */
   …
   }
}
```

Création d'une instance de l'objet simpleClass :

SimpleClass sc = new SimpleClass() ;

Invoquer la méthode sc.safeMethod nécessite de posséder le verrou sur l'instance sc. Si blocage la tâche appelante est placée dans l' *entry set* du verrou de cet objet.

Wait() et notify() sont identique à wait() et signal() du moniteur.

Nottify-All() permet de réveiller toutes les tâches en attente.

```
final class boundedBuffer {
        private int fullSlots=0 ; private int capacity = 0 ;
        private int[ ] buffer = null ; private int in =0, out =0 ;
        public boundedBuffer(int buffer Capacity {
                capacity = bufferCapacity ; buffer = new int[capacity] ;
        }
        public synchronized void deposid (int value) {
            while (fullSlots == capacity) // on suppose pas d'interruptions
                try { wait() ;} catch (InterruptedException ex) {}
            buffer[in] = value ;
            in = (in+1) % capacity ;
            if (fullSlots ++ == 0)
                notifyAll() ; // des consommateurs peuvent être en attente
                                de « pas vide »
        }
```



```
    public synchronized int withdraw () {
            int value = 0 ;
            while (fullSlots == 0)
               try { wait() ; } catch (InterruptedEcxception ex ) {}
            value = buffer[out] ;
            out = (out+1)%capacity ;
            if (fullSlots -- == capacity)
              notifyAll() ; //  des consommateurs peuvent être en attente
                               de « pas plein »
              return value ;
            }
    }

    ….
    BoundedBuffer bb ;
    bb.deposit(…) ;              // exécuté par les producteurs
    bb.withdraw(..) ;            //exécuté par les consommateurs
```