

# Communication et synchronisation en Ada, en C/Posix et en Java 1.5

*Samia Bouzefrane*

Maître de Conférences

CEDRIC –CNAM

samia.bouzefrane@cnam.fr  
<http://cedric.cnam.fr/~bouzefra>

MCours.com

## Sommaire

- **Synchronisation en Ada**
  - Présentation des objets protégés d'Ada
  - Sémantique de base
  - Exemple d'utilisation
  - Présentation du concept de Rendez-Vous en Ada
  - Exemple d'utilisation
- **Synchronisation en C/Posix**
  - Les sémaphores Posix
  - Les variables condition Posix
  - Exemple d'utilisation
- **Synchronisation à partir de Java 1.5**
  - Présentation des moniteurs Java
  - Les variables condition Java 1.5
  - Exemple d'utilisation

# Synchronisation en ADA

## Rappel du concept de moniteur

- **Concept** proposé par **Hoare** en 1974 pour résoudre le problème de synchronisation.

Type m = moniteur

Début

Déclaration des variables locales;

Déclaration et corps des procédures du moniteur; // **accessibles en exclusion mutuelle**

Initialisation;

Fin

- **Les procédures du moniteur se synchronisent à l'aide de deux primitives :**

**Wait()**

**Signal()**

qui permettent de bloquer ou de réveiller un processus sur une condition.

- **Une condition est une variable qui n'a pas de valeur mais qui est implémentée à l'aide d'une file d'attente.**

- **Syntaxe des primitives :**

**Cond.Wait()** : bloque toujours le processus appelant

**Cond.Signal()** : réveille un processus bloqué dans la file d'attente associée à **Cond.**

## Synchronisation en ADA

**Dans le modèle de synchronisation proposé par Ada :**

- **Il n'existe pas de sémaphores**
- **Deux mécanismes sont proposés :**
  - Les « objets protégés » : proche des moniteurs; permettent l'encapsulation de données privées qui sont accédées par des procédures (lectures/écritures des données du moniteur).
  - L'utilisation de « rendez-vous » : proche des RPC entre une tâche serveur qui accepte le rendez-vous et une tâche client qui appelle le serveur pour ce rendezvous

## Objet protégé

- Un objet protégé définit des données privées ne pouvant être accédées que par les sous-programmes (fonctions, procédures ou entrées) associés à l'objet protégé.
- C'est un élément « passif » : ce sont les tâches qui exécutent le code des sous-programmes ou entrées des objets protégés.
- Syntaxe à respecter :

***protected** NomDeLObjetProtege **is***

*déclaration des fonctions, procédures et entrées de l'objet protégé*

***private***

*déclaration des données privées de l'objet protégé*

***end** NomDeLObjetProtege;*

## Objet protégé: sémantique de base

- La partie privée (« *private* ») définit les données ou contenu de l'objet protégé
- Les fonctions définissent des actions de « lecture » du contenu de l'objet protégé (interdiction de modifier la valeur de celles-ci)
- Les procédures et les entrées définissent des actions de « lecture » et « d'écriture » des données de l'objet protégé (contrairement à une procédure ou à une fonction, l'appel à une entrée peut être bloquant)
- Plusieurs lectures peuvent avoir lieu simultanément
- Une action d'écriture exclut toute autre action (lecture ou écriture)

## Objet protégé: syntaxe/1

Déclaration de l'objet protégé et ses points d'entrée :

```
protected type obj_protege is  
    entry A(..);  
    entry B(..);
```

Définition du corps de l'objet protégé :

```
protected body obj_protege is  
    entry A (..) is  
    begin ..  
    end A;  
  
    entry B (..) is  
    begin ..  
    end B;  
end obj_protege;
```

## Objet protégé: syntaxe/2

Déclaration d'une variable de type objet protégé :

```
Obj : obj_protege;
```

Appel des points d'entrée de l'objet protégé dans un programme :

```
Obj.A(...);
```

```
...
```

```
Obj.B(...);
```

## Objet protégé: Exemple

Écrire un programme qui crée deux tâches qui partagent un buffer (à une case) de telle sorte qu'une tâche est productrice d'un message et que l'autre en est la consommatrice. Par exemple, une tâche génère un nombre et le dépose dans le buffer et l'autre récupère la valeur contenue dans le buffer en vue de l'afficher.

Le buffer sera implanté à l'aide d'un objet protégé.

Définir, dans un paquetage de données, un objet protégé à deux points d'entrée :

- Déposer* (*var : in integer*) appelée par la tâche productrice
- Retirer* (*var : out integer*) appelée par la tâche consommatrice.

## Objet protégé: Prod/Cons

```
-- fichier: objet_buffer.ads

package objet_buffer is

    protected type Buffer is
        entry Deposer(X: in float);
        entry Retirer(X: out float);
        private
            Donnee : float;
            Plein : boolean := False;
        end Buffer;
end objet_buffer;
```

## Objet protégé: Prod/Cons

```
-- fichier : objet_buffer.adb
package body objet_buffer is

protected body Buffer is
  entry Deposer (X: in float) when not Plein is
  begin
    Donnee :=X;
    Plein := True;
  end Deposer;

  entry Retirer (X: out float) when Plein is
  begin
    X:= Donnee;
    Plein := False;
  end Retirer;
end Buffer;

end objet_buffer;
```

## Objet protégé: Prod/Cons

```
-- fichier : prod_cons.adb
with text_io; use text_io;
with objet_buffer; use objet_buffer;
with ada.numerics.float_random; use ada.numerics.float_random;

procedure prod_cons is

package es_reel is new text_io.float_io(float);
use es_reel;

N: constant :=3;
Buf: Buffer;

task type Producteur;
task type Consommateur;
```

## Objet protégé: Prod/Cons

```
-- suite de prod_cons.adb
task body Producteur is
G: generator;
mess : float;
    begin
        reset(G);
        mess := Random(G);
        -- produire une valeur
        Buf.Deposer(mess);
        put("Producteur "); put(mess);
        put(" depose ");
        put(mess);
        put_line("--fin Prod");
end Producteur;

task body Consommateur is
c: Float;
    begin
        Buf.Retirer(c);
        put ("Consommateur");
        put(" vient de lire ");
        put(c);
        put_line("--fin Cons");
end Consommateur;
```

## Objet protégé: Prod/Cons

```
-- fin de prod_cons.adb  
Prod : array (1..N) of Producteur;  
Cons: array (1..N) of Consommateur;  
  
begin  
    null;  
end prod_cons;
```

### Exécution:

```
$gcc objet_buffer.adb  
$gcc prod_cons.adb  
$./prod_cons  
...
```

## Le concept de Rendez-Vous en ADA/1

Deux tâches  $T$  et  $L$  peuvent s'attendre mutuellement en vue de réaliser une action : on dit qu'elles réalisent un RDV.

La tâche  $T$  définit alors un point de RDV *proc\_rdv()*:

```
--spécification  
task T is  
    entry proc_rdv(...);  
end T;
```

## Le concept de Rendez-Vous en ADA/2

La tâche *T* définit aussi dans son corps la procédure à exécuter lors du RDV :

```
-- acceptation du RDV
task body T is
....
begin
    instructions;
    accept proc_rdv(...) do -- définit des paramètres
    -- formels comme dans une procédure
        instructions;
    end proc_rdv;
    instructions;
end T;
```

## Le concept de Rendez-Vous en ADA/3

La tâche *L* fait appel à la procédure de RDV de la tâche *T* :

```
-- la tâche L qui sera en RDV avec T
task body L is
....
begin
    instructions;
    T.proc_rdv(...); -- demande de RDV
    instructions;
end L;
```

L'appel d'un RDV permet en général de communiquer des données produites par une tâche et récupérées par la tâche qui définit le RDV.

## Le Rendez-Vous en ADA: exemple/1

Écrire un programme qui crée deux tâches qui doivent se synchroniser par rendez-vous en vue de convertir des caractères. La première tâche sera productrice d'un caractère qu'elle doit communiquer à l'autre tâche afin que cette dernière le convertisse en majuscule.

Le point de RDV est représenté à l'aide de la procédure *recoit()*.

La tâche consommatrice doit définir un point d'entrée avec le nom de cette procédure.

```
entry recoit(c: in character);
```

## Le Rendez-Vous en ADA: exemple/2

```
procedure conversion is
  -- rdv du producteur et du consommateur
  -- en vue de produire un caractère converti en majuscule par
  -- le consommateur

  function majuscule (c: character) return character is
  begin
    if c>='a' and c<='z' then
      return
character'val(character'pos(c)-character'pos('a')+character'pos('A'));
      else return c;
    end if;
  end majuscule;
```

## Le Rendez-Vous en ADA: exemple/3

```
-- spécification des taches producteur et consommateur
task producteur;
task consommateur is
    -- vue externe de la tache
    entry recoit(c: in character);
    -- recoit est un point de synchronisation
    -- c est un parametre formel comme dans un sous-pgme
end consommateur;

-- corps de producteur
task body producteur is
    c: character;
    begin
        while not end_of_file(standard_input) loop
            get(c);
            consommateur.recoit(c);
        end loop;
end producteur;
```

## Le Rendez-Vous en ADA: exemple/4

```
-- corps de consommateur
task body consommateur is
    x: character;
    begin
        loop
            accept recoit(c: in character) do
                -- les noms des taches appelantes ne sont pas spécifiées
                x:=c;
                end recoit;
                put(majuscule(x));
            end loop;
end consommateur;

begin
    null;
end conversion;
```

## Synchronisation en C / Posix

## Mécanismes étudiés

- ♦ **Mutex Posix (sémaphore binaire)**
- ♦ **Sémaphores Posix**
- ♦ **Moniteurs Posix : variables conditionnelles associées aux Mutex**
- ♦ **Nous n'abordons pas**
  - l'utilisation de tubes ("pipe")
  - l'utilisation de « sockets » (API TCP)
  - l'utilisation des sémaphores à la Unix (tableau de sémaphores)  
(voir Livre de Systèmes d'exploitation- exercices en C/Posix et Java,  
de Samia Bouzefrane, édition Dunod 2003)

## Gestion des Mutex

- Un « **Mutex** » est un **sémaphore binaire** pouvant prendre un des deux états
- "**lock**" (verrouillé) ou "**unlock**" (déverrouillé): valeur de sémaphore 1 ou 0
- Un « **Mutex** » ne peut être partagé que par des threads d'un même processus
- Un « **Mutex** » ne peut être verrouillé que par une seule thread à la fois.
- Une thread qui tente de verrouiller un « **Mutex** » déjà verrouillé est suspendu jusqu'à ce que le « **Mutex** » soit déverrouillé.

## Déclaration et initialisation d'un Mutex

- Un mutex est une variable de type « `thread_mutex_t` »
- Il existe une constante `PTHREAD_MUTEX_INITIALIZER` de ce type permettant une déclaration avec initialisation statique du mutex (avec les valeurs de comportement par défaut)

```
pthread_mutex_t monMutex = THREAD_MUTEX_INITIALIZER;
```

- Un mutex peut également être initialisé par un appel de la primitive

```
int pthread_mutex_init(pthread_mutex_t *mutex, const  
pthread_mutexattr_t *mutexattr);
```

avec une initialisation par défaut lorsque `mutexattr` vaut `NULL`

```
ex: pthread_mutex_init(&monMutex, NULL);
```

## Prise (verrouillage) d'un mutex

- Un mutex peut être **verrouillé** par la primitive

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Si le mutex est **déverrouillé** il devient **verrouillé**
- Si le mutex est déjà verrouillé par une autre thread la tentative de verrouillage **suspend** l'appelant jusqu'à ce que le mutex soit déverrouillé.

## Relâchement (déverrouillage) d'un mutex

- Un mutex peut être déverrouillé par la primitive

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Si le mutex est déjà déverrouillé, cet appel n'a aucun effet (comportement par défaut)
- Si le mutex est verrouillé, une des threads en attente obtient le mutex (qui reprend alors l'état verrouillé) et cette thread redevient active (elle n'est plus bloquée)
- L'opération est toujours *non bloquante pour l'appelant*

## Exemple d'utilisation de mutex

```
#define N      10  /* Nb de cases du tampon */  
  
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;  
int tampon[N];
```

```
void Deposer(int m){  
pthread_mutex_lock(&mutex);  
    // déposer m dans tampon;  
    .....
```

```
pthread_mutex_unlock(&mutex);  
}
```

.....

On verrouille le Mutex : accès exclusif

On déverrouille le Mutex

## Gestion des sémaphores Posix

- Un sémaphore Posix est un sémaphore à compte pouvant être partagé par plusieurs threads de plusieurs processus
- Le **compteur** associé à un sémaphore peut donc prendre des valeurs plus grande que 1 (contrairement à un mutex)
- La prise d'un sémaphore dont le compteur est négatif ou nul **bloque** l'appelant.
- Il ne faut pas confondre les sémaphores Posix avec les sémaphores Unix qui sont en fait des tableaux de sémaphores (non étudiés ici).

## Rappel du concept de sémaphore

- **Un sémaphore  $s$  :**  
{**Val(s)**: valeur qui doit toujours être initialisée,  
**File(s)**: file d'attente qui va contenir les processus bloqués sur ce sémaphore }
- **La valeur initiale d'un sémaphore ne doit jamais être négative.**

### Primitive P(s):

Debut

$Val(s) = Val(s) - 1;$

Si  $Val(s) < 0$  Alors Mettre le processus actif dans la file File(s);

Fin

### Primitive V(s):

Debut

$Val(s) = Val(s) + 1;$

Si  $Val(s) \leq 0$  Alors /\* il y a au moins un processus bloqué dans File(s) \*/  
Sortir un processus de la file File(s);

Fin

## Création / Initialisation d'un sémaphore Posix

- Les prototypes des fonctions et les types sont définis dans « **semaphore.h** »
- Un sémaphore est une variable de type « **sem\_t** »
- Il est initialisé par un appel à la primitive :

```
int sem_init(sem_t *sem, int pshared, unsigned int valeur);
```

- si "**pshared**" vaut 0 le sémaphore ne peut pas être partagé entre threads de différents processus (partage uniquement au sein d'un même processus)
- **valeur** définit la valeur initiale de ce sémaphore (positive ou nulle)

## Prise / Relâchement d'un sémaphore Posix

- Les deux opérations P et V sont implémentées par

```
P : int sem_wait(sem_t *sem);  
V : int sem_post(sem_t *sem);
```

avec les mêmes comportements que les primitives génériques P et V

- Il existe également une version non bloquante de la primitive P :

```
int sem_trywait(sem_t *sem);
```

qui retourne 0 lorsque la prise est possible (et non bloquante) et qui retourne **EAGAIN** sinon (dans le cas où l'appel normal serait bloquant)

## Lecture de la valeur d'un sémaphore Posix

- La primitive

```
int sem_getvalue(sem_t *sem, int *sval);
```

qui retourne dans `sval` la valeur courante du sémaphore `sem`

- La primitive

```
int sem_destroy(sem_t *sem);
```

qui permet de libérer les ressources associées au sémaphore `sem`

## Sémaphores vs Mutex

- Les Mutex constituent un cas particulier de sémaphores Posix.
- Les Mutex correspondent à des sémaphores binaires.
  - **Avantages de Mutex**
    - Les primitives de manipulation de Mutex servent uniquement à implémenter une exclusion mutuelle.
    - Les primitives de manipulation de Mutex sont faciles à utiliser.
  - **Avantages sémaphores Posix**
    - Les sémaphores Posix implémentent n'importe quel type de synchronisation entre les threads d'un même processus (en plus de l'exclusion mutuelle).

## Exemple du Prod/Cons avec les Sémaphores Posix

```
/* prodconsThread.c avec des threads*/  
#include <stdio.h>  
#include <unistd.h>  
#include <semaphore.h>  
#include <fcntl.h>  
  
#define Ncases 10 /* nbr de cases du tampon */  
  
int Tampon[Ncases]; /* Tampon a N cases*/  
sem_t Snvide, Snplein; /* les sémaphores */
```

## Exemple du Prod/Cons /Suite

```
void *Producteur(void) {
    int i, queue=0, MessProd;

    srand(pthread_self());

    for(i=0; i<20; i++){
        sleep(rand()%3); /* fabrique le message */
        MessProd = rand() % 10000;
        printf("Product %d\n",MessProd);
        sem_wait(&Snvide);
        Tampon[queue]=MessProd;
        sem_post(&Snplein);
        queue=(queue+1)%Ncases;
    }
    pthread_exit (0);
}
```

## Exemple du Prod/Cons /Suite

```
void *Consommateur(void) {
    int tete=0, MessCons, i;

    srand(pthread_self());
    for(i=0; i<20; i++){
        sem_wait(&Snplein);
        MessCons = Tampon[tete];
        sem_post(&Snvide);
        tete=(tete+1)%Ncases;
        printf("\t\tConsomm  %d \n",MessCons);
        sleep(rand()%3); /* traite le message */
    }

    pthread_exit (0);
}
```

## Exemple du Prod/Cons (Fin)

```
int    main(void) {
        pthread_t th1, th2;

        /* creation et initialisation des semaphores */
        sem_init(&Snvide, 0, Ncases);
        sem_init(&Snplein, 0, 0);

        /* creation des threads */
        pthread_create (&th1, 0, Producteur, NULL);
        pthread_create (&th2, 0, Consommateur, NULL);

        /* attente de terminaison */

        pthread_join (th1, NULL);
        pthread_join (th2, NULL);

        /* suppression des semaphores */
        sem_destroy(&Snplein);
        sem_destroy(&Snvide);
    return (0);
}
```

## Exécution du Prod/Cons

```
$gcc prodconsThread.c -o prodcons -lpthread
$./prodcons
Product 2100
Product 3250

        Consomm  2100
Product 2540
        Consomm  3250
        Consomm  2540

...
```

## Rappel du concept de moniteur

- **Concept** proposé par **Hoare** en 1974 pour résoudre le problème de synchronisation.

Type m = moniteur

Début

Déclaration des variables locales;

Déclaration et corps des procédures du moniteur; // **accessibles en exclusion mutuelle**

Initialisation;

Fin

- **Les procédures du moniteur se synchronisent à l'aide de deux primitives :**

**Wait()**

**Signal()**

qui permettent de bloquer ou de réveiller un processus sur une condition.

- **Une condition est une variable qui n'a pas de valeur mais qui est implémentée à l'aide d'une file d'attente.**

- **Syntaxe des primitives :**

**Cond.Wait()** : bloque toujours le processus appelant

**Cond.Signal()** : réveille un processus bloqué dans la file d'attente associée à **Cond.**

## Moniteurs Posix

- **Un moniteur Posix est l'association**

- **d'un mutex** ( type `pthread_mutex_t` ) qui sert à protéger la partie de code où l'on teste les conditions de progression
- **et d'une variable condition** ( type `pthread_cond_t` ) qui sert de point de signalisation :

- on se met en attente sur cette variable par la primitive :

```
pthread_cond_wait(&laVariableCondition,&leMutex);
```

- on est réveillé sur cette variable avec la primitive :

```
pthread_cond_signal(&laVariableCondition);
```

## Schéma d'utilisation

- Soit la condition de progression C,
- Le schéma d'utilisation des moniteurs Posix est le suivant :

```
pthread_mutex_lock (&leMutex);  
évaluer C;  
while ( ! C ) {  
    pthread_cond_wait(&laVariableCondition,&leMutex);  
    ré-évaluer C si nécessaire  
}  
  
Faire le travail;  
pthread_mutex_unlock(&leMutex);
```

## Exemple du Prod/Cons avec les moniteurs Posix

```
#include <pthread.h>

/* définition du tampon */
#define N      10 /* Nb de cases du tampon */
#define NbMess 20 /* Nb de messages échangés */
int NbPleins=0;
int tete=0, queue=0;
int tampon[N];

/* définition des conditions et du mutex */
pthread_cond_t vide;
pthread_cond_t plein;
pthread_mutex_t mutex;

pthread_t tid[2];
```

## Exemple (suite)

```
void Deposer(int m){  
    pthread_mutex_lock(&mutex);  
    if(NbPleins == N) pthread_cond_wait(&plein, &mutex);  
    tampon[queue]=m;  
    queue=(queue+1)%N;  
    NbPleins++;  
    pthread_cond_signal(&vide);  
    pthread_mutex_unlock(&mutex);  
}
```

```
int Prelever(void){  
    int m;  
    pthread_mutex_lock(&mutex);  
    if(NbPleins ==0) pthread_cond_wait(&vide, &mutex);  
    m=tampon[tete];  
    tete=(tete+1)%N;  
    NbPleins--;  
    pthread_cond_signal(&plein);  
    pthread_mutex_unlock(&mutex);  
    return m;  
}
```

## Exemple (suite)

```
void * Prod(void * k)                /****** PRODUCTEUR */
{
int i;
int mess;
 srand(pthread_self());
for(i=0;i<=NbMess; i++){
    usleep(rand()%10000); /* fabrication du message */
    mess=rand()%1000;
     Deposer(mess);
    printf("Mess depose: %d\n",mess);
}
}
void * Cons(void * k)                /****** CONSOMMATEUR */
{
int i;
int mess;
 srand(pthread_self());
for(i=0;i<=NbMess; i++){
     mess=Prelever();
    printf("\tMess preleve: %d\n",mess);
    usleep(rand()%1000000); /* traitement du message */
}
}
46 }
```

## Exemple (fin)

```
void main()                                /* M A I N */
{
  int i,  num;
  pthread_mutex_init(&mutex,0);
  pthread_cond_init(&vide,0);
  pthread_cond_init(&plein,0);

  /* creation des threads */
  pthread_create(tid, 0, (void * (*)()) Prod, NULL);
  pthread_create(tid+1, 0, (void * (*)()) Cons, NULL);

  // attente de la fin des threads
  pthread_join(tid[0],NULL);
  pthread_join(tid[1],NULL);

  // libération des ressources
  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&vide);
  pthread_cond_destroy(&plein);

  exit(0);
}
```

# Synchronisation en Java

## Les moniteurs Java/1

- **Jusqu'à la version 1.4, seuls les moniteurs étaient proposés en Java**
  - Les sémaphores pouvaient être implémentés à l'aide des moniteurs Java
  
- **A partir de la version 1.5, les sémaphores et les verrous sont proposés**

## Les moniteurs Java/2

- Le concept de moniteur est implémenté dans la MVJ de la manière suivante :
  - Les données du moniteur doivent être déclarées avec le mot clé **private** pour que seules les méthodes du moniteur accèdent à ces données,
  - Les méthodes (ou procédures d'entrée) du moniteur doivent être déclarées avec le mot clé **synchronized** pour qu'elles puissent s'exécuter en exclusion mutuelle,
  
- La classe **Object** fournit les méthodes **wait()** et **notify()** pour la synchronisation des threads.
  - Jusqu'à la version 1.4, la notion de variable condition n'existait pas et les méthodes **wait()** et **notify()** étaient associées directement à l'objet moniteur.
  - A partir de la version 1.5, la notion de variable condition a été rajoutée ainsi que des méthodes matérialisant les verrous.

## Exemple du Prod/Conso en Java 1.4

```
interface ProdConsInterface {
    public void Deposer(int m) throws InterruptedException ;
    public int Prelever() throws InterruptedException ;
}

class ProdConsMonitor implements ProdConsInterface {
    private int NbPleins=0, tete=0, queue=0 ;
    private int N ;
    private int tampon[];
    ProdConsMonitor (int n) {
        N=n ;
        tampon = new int [N] ;
    }
}
```

## Exemple (suite)

```
public synchronized void Deposer (int m) throws InterruptedException {
    while (NbPleins == N) { wait(); }
    tampon[queue] =m ;
    queue = (queue +1)% N;
    NbPleins ++;
    System.out.println(Thread.currentThread().getName() +
" vient de produire " + m) ;
    notifyAll();
}
public synchronized int Prelever () throws InterruptedException {
    while (NbPleins == 0 ) {wait(); }
    int m= tampon[tete];
    tete = (tete + 1)% N;
    NbPleins --;
    notifyAll();
    System.out.println(Thread.currentThread().getName() +
" vient de consommer " + m) ;
    return m ;
}

} // fin de ProdConsMonitor
```

## Exemple (suite)

**La classe Producteur :**

```
class Producteur extends Thread {  
    ProdConsMonitor Mo;  
    Producteur ( ProdConsMonitor Mo ) {  
        this.Mo=Mo;  
    }  
    public void run() {  
        try {  
            while (true) {  
                int m= (int)(1000*Math.random());  
                Mo.Deposer(m) ;  
                Thread.sleep((int)(1000*Math.random()));  
            }  
        } catch(InterruptedException e) {}  
    }  
} // fin run  
} // fin de la classe Producteur
```

## Exemple (suite)

La classe **Consommateur** :

```
class Consommateur extends Thread {  
  
    ProdConsMonitor Mo;  
    Consommateur (ProdConsMonitor Mo) {  
        this.Mo=Mo;  
    }  
    public void run() {  
        try {  
            while (true) {  
                int m = Mo.Prelever() ;  
                Thread.sleep((int)(1000*Math.random()));  
            }  
        } catch(InterruptedException e) {}  
    } // fin run  
} // fin de la classe Consommateur
```

## Exemple (fin)

**La classe main :**

```
// A exécuter avec un argument = taille du tampon
class ProdConsMain {
    public static void main(String argv[]) {
        int N= Integer.parseInt(argv[0]);
        ProdConsMonitor pc = new ProdConsMonitor(N) ;

        Producteur producteur = new Producteur (pc) ;
        Consommateur consommateur = new Consommateur (pc) ;

        producteur.setName("Producteur") ;
        producteur.start() ;
        consommateur.setName("Consommateur") ;
        consommateur.start() ;
        producteur.join() ;
        consommateur.join() ;
    }
}
```

## La synchronisation avec Java 1.5 et plus

### ➤ Inconvénients des moniteurs Java $\leq 1.4$ :

- On ne peut pas bloquer une thread sur une condition particulière
- Par conséquent, on ne peut pas réveiller une thread particulière car toutes les threads bloquées sont dans la file d'accès au moniteur
- Pas de condition → pas de file d'attente dans le moniteur

➤ **Java Community Process a proposé à travers l'activité (JCP, JSR 166, 2002)** un ensemble d'utilitaires relatifs à la concurrence qui sont rajoutés à partir de la version 1.5 de Java.

### ➤ Parmi les paquetages rajoutés, il y a :

- **`java.util.concurrent.locks`** : qui fournit des mécanismes de verrouillage (verrous de lect/écrit, variables conditions)
- **`java.util.concurrent`** : qui fournit entre autres une classe **Semaphore** qui permet la synchronisation à l'aide des sémaphores

## Les sémaphores en Java 1.5 et plus

### ➤ Paquetage `java.util.concurrent`

```
public class Semaphore extends Object implements Serializable
```

### ➤ Constructeur :

**Semaphore(int permits)** : Crée un Sémaphore avec une valeur initiale (nb d'appels non bloquants)

**Semaphore(int permits, boolean fair)** : Crée un Sémaphore avec une valeur initiale (nb d'appels non bloquants). `fair = true` pour garantir une gestion FIFO des processus en cas de conflit.

### ➤ Méthodes :

▪ **void acquire ()** : demande une permission (un ticket), la thread appelante se bloque si pas de ticket ou bien si elle est interrompue.

▪ **void acquire (int permits)** : demande un certain nombre de tickets, la thread appelante se bloque si pas de ticket ou bien si elle est interrompue.

▪ **void release ()** : libère un ticket.

▪ **void release (int permits)** : libère un certain nombre de tickets.  
etc.

## Exemple d'utilisation des sémaphores

```
Semaphore s = new Semaphore (1, true);
```

```
s.acquire();
```

```
Accès à la section critique;
```

```
s.release();
```

## Les variables condition en Java 1.5 et plus

### ➤ Paquetage `java.util.concurrent.locks`

`public interface Condition` : remplace l'utilisation des méthodes du moniteur.

### Méthodes :

- `void await ()` : provoque le blocage de la thread appelante jusqu'à réception d'un signal ou bien d'une interruption.
- `void await (long time, TimeUnit unit)` : provoque le blocage de la thread appelante jusqu'à réception d'un signal, d'une interruption ou bien le temps `unit` est dépassé.
- `void signal ()` : réveille une thread bloquée.
- `void signalAll ()` : réveille toutes les threads bloquées.

etc.

## Les verrous en Java 1.5 et plus

### ➤ Paquetage `java.util.concurrent.locks`

public interface **Lock** : remplace l'utilisation des méthodes `synchronized` (accessibles en exclusion mutuelle).

### Méthodes :

- void **lock** () : la thread appelante obtient un verrou.
- void **lockInterruptibly** () : la thread appelante obtient un verrou et le garde jusqu'à son interruption.
- Condition **newCondition** () : retourne une nouvelle instance `Condition` à utiliser avec un verrou de type `Lock`.
- void **unlock** () : restitue le verrou.

etc.

## Les verrous en Java 1.5 et plus

### ➤ Paquetage `java.util.concurrent.locks`

`public class ReentrantLock` extends `Object` implements `Lock`, `Serializable` : est un verrou d'exclusion mutuelle.

### Méthodes :

- void `lock()` : la thread appelante obtient un verrou.
- void `lockInterruptibly()` : la thread appelante obtient un verrou et le garde jusqu'à son interruption.
- `ConditionObject newCondition()` : retourne une nouvelle variable `Condition` liée à une instance de `Lock`.
- void `unlock()` : restitue le verrou.

```
class X {  
private final ReentrantLock verrou = new ReentrantLock();  
// ...  
public void m() {  
verrou.lock(); // bloque jusqu'à satisfaction de la condition  
try { // ... Corps de la méthode  
} finally { verrou.unlock() } } }
```

## Exemple du Prod/Conso en Java 1.5 et plus

```
import java.util.concurrent.locks.*;
interface ProdConsInterface {
    public void Deposer(int m) throws InterruptedException ;
    public int Prelever() throws InterruptedException ;
}

class ProdConsMonitor implements ProdConsInterface {
    final Lock verrou = new ReentrantLock();
    final Condition Plein = verrou.newCondition();
    final Condition Vide = verrou.newCondition();
    int tampon[];

    int queue,tete, N, NbPleins=0,NbVides=0;

    ProdConsMonitor (int n) {
        N=n ;
        tampon = new int [N] ;
    }
}
```

## Exemple (suite)

```
public void Deposer (int m) throws InterruptedException {
    verrou.lock();
    try {
        if (NbPleins == N) Plein.await();

        tampon[queue] =m ;
        queue = (queue +1)% N;
        NbPleins ++;
        System.out.println(Thread.currentThread().getName() +
            " vient de produire " + m) ;
        Vide.signal();

    } finally {
        verrou.unlock();
    }
}
```

## Exemple (suite)

```
public int Prelever () throws InterruptedException {
    verrou.lock();
    try {
        if (NbPleins == 0 ) Vide.await();
        int m= tampon[tete];
        tete = (tete + 1)% N;
        NbPleins --;
        Plein.signal();
        System.out.println(Thread.currentThread().getName() +
            " vient de consommer " + m) ;
        return m ;

    } finally {
        verrou.unlock();
    }
}

} // fin de ProdConsMonitor
```

## Exemple (suite)

La classe Producteur :

```
class Producteur extends Thread {
    ProdConsMonitor Mo;
    Producteur ( ProdConsMonitor Mo ) {
        this.Mo=Mo;
    }
    public void run() {
        try {
            while (true) {
                int m= (int)(1000*Math.random());
                Mo.Deposer(m) ;
                Thread.sleep((int)(1000*Math.random()));
            }
        } catch(InterruptedException e) {}
    }
} // fin run
} // fin de la classe Producteur
```

## Exemple (suite)

La classe **Consommateur** :

```
class Consommateur extends Thread {  
  
    ProdConsMonitor Mo;  
    Consommateur (ProdConsMonitor Mo) {  
        this.Mo=Mo;  
    }  
    public void run() {  
        try {  
            while (true) {  
                int m = Mo.Prelever() ;  
                Thread.sleep((int)(1000*Math.random()));  
            }  
        } catch(InterruptedException e) {}  
    } // fin run  
} // fin de la classe Consommateur
```

## Exemple

```
//La classe main : à exécuter avec un argument = taille du tampon
class ProdConsMain {
    public static void main(String argv[]) {
        int N= 0 ;
        try{ N=Integer.parseInt(argv[0] );}
        catch(ArrayIndexOutOfBoundsException exp){
            System.out.println("USAGE : java ProdConsMain [taille du tampon]");
            System.exit(0);
        }

        ProdConsMonitor pc = new ProdConsMonitor(N) ;
        Producteur producteur = new Producteur (pc) ;
        Consommateur consommateur = new Consommateur (pc) ;

        producteur.setName("Producteur") ;
        producteur.start() ;
        consommateur.setName("Consommateur") ;
        consommateur.start() ;
        producteur.join(); consommateur.join();
    }
}
```

## Exemple (fin)

### Exemple d'exécution:

```
D:\java\jdk1.5\bin\java ProdConsMain 3
Producteur vient de produire 515
Consommateur vient de consommer 515
Producteur vient de produire 482
Consommateur vient de consommer 482
Producteur vient de produire 610
Consommateur vient de consommer 610
Producteur vient de produire 173
Consommateur vient de consommer 173
Producteur vient de produire 360
Producteur vient de produire 192
Consommateur vient de consommer 360
Producteur vient de produire 15
Producteur vient de produire 402
Consommateur vient de consommer 192
...
```

## Conclusion

- **Ada n'offre pas de sémaphore mais propose :**
  - les objets protégés qui ont la sémantique des moniteurs et qui se manipulent très simplement
  - offre aussi le concept de Rendez-Vous inexistant dans d'autres langages.
  
- **Posix propose plusieurs mécanismes**
  - **Mutex** : pour les sections critiques
  - **Sémaphores** : pour la synchronisation entre threads en général
  - **Variables condition** : pour l'utilisation de moniteur
  
- **Java dans sa version 1.5 et plus propose :**
  - une classe **Object**: pour l'utilisation de moniteurs sans variables condition
  - une classe **Sémaphore** : pour la synchronisation entre threads en général
  - une classe **Condition** : pour l'utilisation de moniteur avec des variables condition

## Références

Jean-François Peyre, supports de cours sur l'informatique industrielle-systèmes temps réel, CNAM(Paris).

Samia Bouzefrane, LES SYSTEMES D'EXPLOITATION: COURS ET EXERCICES CORRIGES UNIX, LINUX et WINDOWS XP avec C et JAVA (566 pages), Dunod Editeur, Octobre 2003, ISBN : 2 10 007 189 0.

MCours.com