Moniteurs

— notations, sémantique, applications —

Introduction

Les sémaphores restent un mécanisme de bas-niveau :

- manque de structure : imbrication des opérations P et V à contrôler ;
- un sémaphore est global à un ensemble de processus ;
- utilisé pour exprimer l'exclusion mutuelle ET l'attente sur une condition.

Caractéristiques principales du moniteur

- encapsulation et abstraction des données,
- exclusion mutuelle implicite dans le moniteur,
- entité passive, donc programme = processus actifs + moniteurs passifs,
- variables spéciales pour l'attente sur une condition.

Historique

- Dijkstra 1972 introduit la notion de "secrétaire"
- Hansen 1972 propose une langage avec "*shared class*"
- Hoare 1973 propose la nom de "*monitor*" et clarifie sa sémantique

Notations et sémantique

Remarque : pas de moniteurs en Promela, notation ad-hoc

Une interface et une implémentation

```
monitor CS {
  int n = 0; //  variables permanentes privées initialisées
  // Invariant : n >= 0
  operation inc() { // opération sous le contrôle du moniteur
    int temp;
    temp = n; n = temp + 1;
  } }
```

- variables permanentes = ressources contrôlées :
  - invisibles de l'extérieur ;
  - initialisées avant tout appel d'opération.
- opération = action sur les ressources :
  - les seules visibles de l'extérieur : CS.inc();
  - elles n'accèdent pas les variables externes.
- invariant du moniteur = décrit l'état des ressources quand aucun processus ne les accède :
  - l'initialisation doit l'établir,
  - chaque opération doit le maintenir.

Moyens de synchronisation (1)

Exclusion mutuelle des opérations : par définition, un seul processus peut exécuter à un moment donné les opérations du moniteur.

Métaphore de la pièce-moniteur :

- ressources stockés dans une pièce,
- plusieurs opérations possibles dans cette pièce,
- le moniteur assure qu'une seule personne entre dans la pièce.

Autrement dit : l'appel d'une opération pose un “lock” implicite sur le moniteur.

... mais aucune politique est imposée sur l'attente pour l'entrée dans un moniteur, donc famine possible.

En général :

- moniteur = entité statique, objet du type déclaré
- l'exclusion mutuelle est faite au niveau de chaque objet moniteur

Moyens de synchronisation (2)

Sur condition :

1. explicite : variables spéciales appelées variables de condition, locales au moniteur.
2. implicite : attente sur une condition booléenne contrôlée par le moniteur.

Variables de condition :

- opérations : waitC, signalC, emptyC
- sont indépendantes des moniteurs ; en effet, elles sont fournies par Pthread, mais pas de base en Java.

Exemple : simulation d'un sémaphore

```
monitor Sem { // sémaphore généralisé
  int s = k;   // Invariant : s >= 0
  condition notZero;
  operation wait () {
    if (s==0) waitC(notZero);
    s = s-1;          }
  operation signal () {
    s = s+1;
    signalC(notZero); }
}
```

Variables de condition

- Déclararation “**condition** cv” ; une file (FIFO) est crée, initialement vide, pour les processus en attente à cette condition.
- Test file vide “**emptyC**(cv)”.
- Attente condition “**waitC**(cv)” :

```
waitC(cv) { // exécutée par le processus p
  ajout de p dans la file de cv
  p.state = bloqué
  moniteur.lock = libre
}
```

- Signale condition “**signalC**(cv)” :

```
signalC(cv) { // exécutée par le processus p
  if (file non vide) {
    enlève q processus en tête de file
    q.state = prêt
  }
}
```

Comparaison variables de condition et sémaphores

Sémaphore	Variable de condition
wait peut ou pas bloquer	waitC bloque toujours
signal a toujours un effet	signalC n'a pas d'effet si file vide
signal débloque un processus arbitrairement	signalC débloque le processus en tête
un processus débloqué par signal peut s'exécuter immédiatement	un processus débloqué par signalC doit attendre que le signalant quitte le moniteur

Exercice : en utilisant le diagramme des états, prouver que l'impléméntation du sémaphore assure l'exclusion mutuelle.

Exemple : Producteur-Consommateur avec tampon fini

```
monitor PC {
  bufferTyp buf = bufferEmpty() ; // tampon encapsulé
  condition notEmpty; // condition : on peut consommer
  condition notFull; // condition: on peut produire
  operation put(dataTyp d) { // appelée par le producteur
    if (buf.is_full()) waitC(notFull);
    append(d,buf);
    signalC(notEmpty);
  }
  operation take(dataTyp d) { // appelée par le consommateur
    int d;
    if (buf.is_empty()) waitC(notEmpty);
    d = pop(buf);
    signalC(notFull); // (*) qui a le contrôle ici ??
    return d;
  }
}
```

- encapsulation complète du tampon
- exclusion producteur/consommateur
- après le signalC à (*), qui se trouve dans le moniteur ?

Politiques d'ordonnancement après signalC

Problème : après un signalC, le processus signalant et le processus signalé se retrouvent à deux dans le moniteur !!

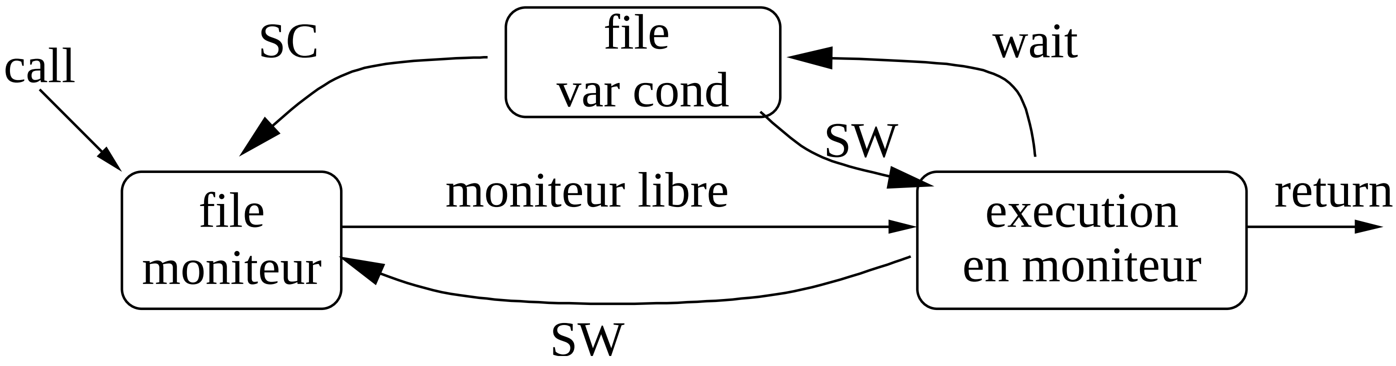
Solution : donner le moniteur à UN seul de ce processus.

Soient :

- E la priorité des processus en attente d'entrer dans le moniteur
- W la priorité du processus debloqué par le signalC
- S la priorité du processus qui effectue signalC

Deux politiques d'ordonnancement utilisées :

- Signale et continue (SC) : quand $E = W < S$, donc le signalant reste dans le moniteur et le signalé est ordonnancé pour l'entrée au moniteur
- Signale et attend (SW) : quand $E < S < W$, le signalant cède le moniteur et le signalé est exécuté.



Politique SW :

- plus ancienne
- simplifie la decription du processus signalé
- peut limiter la concurrence en retardant inutilement le signalant

Politique SC :

- plus simple pour le signalant
- choisie par Java et Pthreads
- nécessite de retester la condition dans le signalé :

```
operation put(dataTyp d) { // appelée par le producteur
  while (buf.is_full()) waitC(notFull);
  append(d,buf);
  signalC(notEmpty);
}
```

Implémentation des moniteurs avec des sémaphores

Pour chaque moniteur M ayant la variable conditionnelle cv déclarer comme variables globales :

```
bsem eM = 1; /* Invariant 0<= eM <=1 */
sem sCV = 0; /* Invariant 0<= sCV */
int nCV = 0; /* Invariant 0<= nCV */
```

On implémente les opérations comme suit :

```
/* entree moniteur = appel opération */
   wait(eM);

/* appel waitC(cv) */
   nCV=nCV+1; signal(eM); wait(sCV); wait(eM)

/* appel signalC(cv) */
   if (nCV > 0) { nCV=nCV-1; signal(sCV); }

/* sortie moniteur = fin opération */
   signal(eM);
```

Exemple : Lecteurs et écrivains

Solution avec moniteur pour le contrôle de l'accès

- un lecteur doit exclure un écrivain mais pas un autre lecteur
- un écrivain doit exclure d'autres écrivains et les lecteurs

```
moniteur RW_Control {
  int readers = 0, writers = 0; // données privées
  condition okToRead;   // attente de writers==0
  condition okToWrite; // attente de readers==0 && writers==0
  // I : (readers>0 => writers=0) && (writers<=1) &&
  //     (writers=1 => readers=0) &&
  //     (readers>=0) && (writers>=0)
  operation startRead() {
    while(writers>0)
      waitC(okToRead)
    readers++;
    signalC(okToRead); // politique SC
  }
  operation endRead() {
    readers--;
    if (readers==0) signalC(okToWrite);
  }
```

```
  operation startWrite() {
    while(writers!=0 || readers!=0)
      waitC(okToWrite)
    writers++;
  }
  operation endWrite() {
    writers--;
    if (empty(okToRead)) signalC(okToWrite);
    else signalC(okToRead);
  }
}
```

Exercices :

1. Prouver que I est un invariant.
2. Que peut-on dire de la famine des lecteurs/écrivains ?

Moniteurs en Java

Il n'y a pas de construction spéciale de Java pour les moniteurs, tout objet peut contrôler l'accès à ses champs ou méthodes par son mutex.

L'accès exclusif, après l'acquisition du mutex, est indiqué par le mot clé **synchronized**.

Si un processus détient le mutex d'un objet, il peut appeler d'autres méthodes/portion de code **synchronized** de l'objet.

Un moniteur en Java = un objet dont tous les champs sont privés et toutes les méthodes sont **synchronized**.

Il n'y a pas d'équité garantie sur l'attente à l'acquisition du mutex.

Une seule variable de condition est définie implicitement dans le moniteur :

- wait() : permet de bloquer le processus appelant
- notify() : equivalent de signalC avec politique SC
- notifyAll() : signalC pour tous les processus bloqués sur la variable de condition de l'objet

Producteur-Consommateur en Java

```
class PC {
  final int N = 5;
  int oldest=0, newest=0;
  int count=0;
  int buffer[] = new int[N];
  synchronized void put(int d) {
    while(count==N)
      try { wait(); } catch (InterruptedException e) {}
    buffer[newest]=v;
    newest=(newest+1)%N;
    count++;
    notifyAll();
  }
  synchronized int take() {
    int temp;
    while(count==0)
      try { wait(); } catch (InterruptedException e) {}
    temp=buffer[oldest];
    oldest=(oldest+1)%N;
    count--;
    notifyAll();
    return temp;
  }
```

Attente sur plusieurs conditions

Doit être simulée en Java à l'aide de notifyAll et boucles de test de la condition :

```java
synchronized int method1() {
  while (!c1) wait();
  ...
}
synchronized int method2() {
  while (!c2) wait();
  ...
}
synchronized int method3() {
  ... // rend c1 ou c2 vraie
  notifyAll();
}
```

Bloc d'instructions en exclusion mutuelle

synchronized peut être appliqué à un bloc d'instructions, ce qui équivaut à l'exécuter en exclusion mutuelle avec d'autres instructions synchronisées avec le même objet.

```
Object obj = new Object();
// processus p1
synchronized (obj) { I1 }

// processus p2
synchronized (obj) { I2 }

// I1 est exécuté en exclusion mutuelle avec I2
```