Les moniteurs

Extrait du cours CNAM-ACCOV
Samia Bouzefrane

Maître de Conférences

CEDRIC –CNAM

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

Bilan du contrôle de concurrence par sémaphores

- Le sémaphore est un mécanisme qui permet le blocage et le réveil explicites de processus
- Le sémaphore permet d'associer un nombre d'autorisations d'accès disponibles à un objet partagé
- Le schéma des sémaphores privés permet d'exprimer une condition d'accès propre à un processus

Tout paradigme de la concurrence peut être traité par l'emploi de sémaphores

Bilan du contrôle de concurrence par sémaphores

Les contraintes de la programmation par sémaphores

-respect des spécifications de programmation=>
pas d'accès aux objets partagés incontrôlé (hors protocole)
attention :exceptions, erreurs, trappes…

-respect des spécifications de comportement :
pas de boucle infinie, ni de blocage ou panne pendant l'utilisation des objets partagés

-éventuellement spécifier les règles d'attente : priorité, ancienneté, équité

Le sémaphore est un mécanisme simple et puissant, mais " dangereux" dans un emploi non contrôlé

Bilan du contrôle de concurrence par sémaphores

La modularité nécessaire

- Encapsuler objets partagés et mécanismes de contrôle
- Mécanismes de contrôle associés exclusivement aux méthodes d'accès
- Les méthodes d'accès accessibles uniquement par l'interface du module, l'implantation est cachée

Remarque : difficulté à évaluer le comportement des processus vis-à-vis de propriétés globales : interblocage, famine

Bilan du contrôle de concurrence par sémaphores

Conclusion

Les sémaphores sont utilisés comme un mécanisme de bas niveau pour implanter des structures de contrôle modulaires

Les **moniteurs** spécifiés par Hoare et Brinch Hansen reposent sur les principes suivants :

-Exclusion mutuelle implicite entre les méthodes d'accès
=>file d'attente au module

-Conditions d'accès reposant sur des tests de variables d'état
=>file d'attente par condition d'accès

Les moniteurs

Les moniteurs

- **Les moniteurs proposent une solution de "haut-niveau" pour la protection de données partagées (Hoare 1974)**

- **Ils simplifient la mise en place de sections critiques**

- **Ils sont définis par**
  – des données internes (appelées aussi variables d'état)
  – des primitives d'accès aux moniteurs (points d'entrée)
  – des primitives internes (uniquement accessibles depuis l'intérieur du moniteur)
  – une ou plusieurs files d'attentes

Structure d'un moniteur

Type *m* = **moniteur**
Début
Déclaration des variables locales (ressources partagées);
Déclaration et corps des procédures du moniteur (points d'entrée);
Initialisation des variables locales;
Fin

Les moniteurs: sémantique/1

- **Seul un processus (ou tâche ou thread) peut être actif à un moment donné à l'intérieur du moniteur**

- **La demande d'entrée dans un moniteur (ou d'exécution d'une primitivedu moniteur) sera bloquante tant qu'il y aura un processus actif à l'intérieur du moniteur**

⇒ *L'accès à un moniteur construit donc implicitement une exclusion mutuelle*

Les moniteurs: sémantique/2

- **Lorsqu'un processus actif au sein d'un moniteur ne peut progresser dans son travail (une certaine condition est fausse), il libère l'accès au moniteur avant de se bloquer.**

- **Lorsque des variables internes du moniteur ont changé, le moniteur doit pouvoir « réveiller » un processus bloqué.**

- **Pour cela, il existe deux types de primitives :**
  - *wait* : qui libère l'accès au moniteur, puis bloque le processus appelant sur une condition
  - *signal* : qui réveille sur une condition un des processus en attente à l'intérieur du moniteur (un processus qui a exécuté précédemment un *wait* sur la même condition)

Les variables condition/1

- **Une variable condition** : est une variable
  - qui est définie à l'aide du type *condition;*
  - qui a un identificateur mais,
  - qui n'a **pas de valeur** (contrairement à un sémaphore).

- **Une condition :**
  - ne doit pas être initialisée
  - ne peut être manipulée que par les primitives Wait et Signal.
  - est représentée par une **file d'attente** de processus bloqués sur la même cause;
  - est donc assimilée à sa file d'attente.

Les variables condition/2

- **La primitive *Wait*** bloque **systématiquement** le processus qui l'exécute

- **La primitive *Signal*** réveille un processus de la file d'attente de la condition spécifiée, si cette file d'attente n'est pas vide;
sinon elle ne fait absolument rien.

Les variables condition/3

- ***Syntaxe*** :

  cond.**Wait**;

  cond.**Signal**;

  /* *cond* est la variable de type condition déclarée comme variable locale */

- ***Autre syntaxe*** :

  Wait(*cond*) ;

  Signal(*cond*);

- Un processus réveillé par *Signal* continue son exécution à l'instruction qui suit le *Wait* qui l'a bloqué.

Les moniteurs dans les langages de programmation

•Selon les langages (ou les normes), ces mécanismes peuvent être Implémentés de différentes façons

– méthodes « *wait / notify / notifyAll* » en Java et méthodes « *synchronized* »
– primitives « *pthread_cond_wait / pthread_cond_signal* » en Posix et Variables conditionnelles
– objets protégés en Ada

•La sémantique des réveils peut varier :

– Qui réveille t-on ?
le plus ancien, le plus prioritaire, un choisi au hasard, …
– Quand réveille t-on ?
dès la sortie du moniteur, au prochain ordonnancement, …

Un RDV entre N processus à l'aide des moniteurs

```
Type Rendez_vous = moniteur
        {variables locales }
        Var Nb_arrivés : entier ; Tous_Arrivés : condition ;
        {procédure accessible aux programmes utilisateurs }
        Procedure Entry Arriver ;
        Début
                Nb_arrivés = Nb_arrivés + 1 ;
                Si Nb_arrivés < N Alors Tous_Arrivés.Wait ;
                Tous_Arrivés.Signal ;
        Fin
        Début  {Initialisations }
                Nb_arrivés = 0;
        Fin.
```

Un RDV entre N processus à l'aide des moniteurs

Les programmes des processus s'écrivent alors :

Processus Pi

.....

***Rendez_vous*.Arriver** ; {*Point de rendez-vous: sera bloquant si au moins un processus n'est pas arrivé au point de rendez-vous* }

.....

Producteur-Consommateur à l'aide des moniteurs

```
Type ProducteurConsommateur = moniteur
        {variables locales }
        Var Compte : entier ; Plein, Vide : condition ;
        {procédures accessibles aux programmes utilisateurs }

        Procedure Entry Déposer(message M) ;
        Début
                si Compte=N alors Plein.Wait ;
                dépôt(M);
                Compte=Compte+1;
                si Compte==1 alors Vide.Signal;
        Fin
```

Producteur-Consommateur à l'aide des moniteurs

```
Procedure Entry Retirer(message M) ;
Début
        si Compte=0 alors Vide.Wait ;
        retrait(M);
        Compte=Compte-1;
        si Compte==N-1 alors Plein.Signal;
Fin

Début  {Initialisations }Compte= 0; Fin.
```

Producteur-Consommateur à l'aide des moniteurs

Processus Producteur

```
message M;
        Début
                tant que vrai faire
                 Produire(M);
                 ProducteurConsommateur.déposer(M)
        Fin
```

Processus Consommateur

```
message M;
        Début
                tant que vrai faire
                 ProducteurConsommateur.retirer(M);
                Consommer(M);
        Fin
```

Remarques

-Adapté au cas de plusieurs producteurs, plusieurs consommateurs

Le repas des philosophes à l'aide des moniteurs

```
Type RepasPhilosophes = moniteur
        type Statut =(pense, demande, mange);
        Var Statut état[5];
        Condition  AMoi[5];
         {procédures accessibles aux programmes utilisateurs }
         Procedure Entry Prendre(entier i);
        Début
                état[i]=demande;
                si (état[(i+1)%5]≠mange et état[(i-1)%5]≠mange)
                alors état[i]=mange;
                sinon AMoi[i].Wait; finsi;
        Fin
```

Le repas des philosophes à l'aide des moniteurs

Procedure Entry Rendre(entier i);
Début
état [i]=pense;
{réveil éventuel d'un voisin}
si (état[(i+1)%5]=demande et état[(i+2)%5]≠mange)
alors état[(i +1)%5]=mange; *AMoi[(i+1)%5]*.Signal;
sinon
si (état[(i-1)%5]=demande et état[(i-2)%5]≠mange)
alors état[(i-1)%5]=mange; *AMoi[(i-1)%5]*.Signal ;finsi;
finsi;
Fin

Début {*Initialisations* }
Pour i de 0 à 4 faire état[i]=pense; fait;
Fin

Le repas des philosophes à l'aide des moniteurs

Processus Philosophe i

```
entier i= numero du processus  ;
        Début
        tant que vrai faire
                penser;
                RepasPhilosophes.prendre(i);
                manger;
                RepasPhilosophes.rendre(i);
        fait;

        Fin
```

Remarque Solution sans interblocage, mais risque de famine

Références

Samia Bouzefrane, Les Systèmes 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.

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