Slide 1

5 Moniteurs

Motivation = les sémaphores peuvent être utilisés pour résoudre à peu près n'importe quel problème d'exclusion mutuelle ou synchronisation . . . mais, les sémaphores possèdent certains désavantages:

- Mécanisme de *bas niveau* qui demande une discipline sévère dans la façon dont ils sont utilisés, sous peine d'erreurs: que se passe-t-il si on oublie d'indiquer un appel à V? Ou si on effectue une action P en trop?
- Mécanisme sans localité: un sémaphore doit être connu et accessible par *tous* les processus qui pourraient devoir l'utiliser $\Rightarrow$ doit être une variable globale $\Rightarrow$ tout le programme doit être examiné pour voir où et comment le sémaphore est utilisé
- Le rôle d'une opération P ou V (exclusion mutuelle? synchronisation conditionnelle?) dépend du type de sémaphore, de la façon dont il est initialisé et manipulé par les divers processus $\Rightarrow$ pas explicite

Slide 2

Solution alternative = Moniteur = Forme de *module* qui supporte, à l'aide de deux mécanismes indépendants, l'exclusion mutuelle et la synchronisation conditionnelle

Caractéristique majeure d'un programme avec moniteurs = Composé de deux sortes de modules/processus:

1. Des processus *actifs*
2. Des moniteurs *passifs*

Les interactions et synchronisations entre processus (actifs) se font alors *par l'intermédiaire* de moniteurs (passifs)

Slide 3

5.1 Syntaxe et sémantique

Moniteur = module qui regroupe et encapsule une ressource partagée (données) ainsi que les procédures qui permettent de manipuler cette ressource

Forme générale d'une déclaration de moniteur dans la notation d'Andrews:

```
monitor nom_du_moniteur {
  ... Declarations des variables definissant
      la ressource partagee ...

  ... Initialisation des variables ...

  ... Declarations des procedures permettant
      la manipulation de la ressource ...
}
```

Moniteur $\approx$ type abstrait, mais avec des propriétés d'exclusion mutuelle et de synchronisation lorsque le moniteur est partagé par plusieurs processus concurrents

Slide 4

5.1.1 Exclusion mutuelle

Philosophie des moniteurs = Séparer de façon claire l'exclusion mutuelle de la synchronisation conditionnelle:

- L'exclusion mutuelle est supportée de façon *implicite*: un appel, par un processus, d'une procédure exportée par le moniteur assure que la procédure sera exécutée de façon *exclusive*, c'est-à-dire, au plus un appel d'une procédure du moniteur sera actif à un instant donné
- Les synchronisations doivent être décrites et programmées de façon *explicite* à l'aide de variables de condition (*condition variables*)

En d'autres mots, l'exclusion mutuelle est automatique, sa mise en oeuvre étant assurée par le langage, par la bibliothèque, ou par le système d'exploitation, pas par le programmeur lui-même

Slide 5

5.1.2 Variables de condition

Une variable de condition est utilisée pour suspendre un processus jusqu'à ce qu'une certaine condition devienne vraie

- Déclaration:

```
cond cv;        /* Nouveau type cond. */
```

- Valeur d'une variable de condition = ensemble (plus précisément, file) des processus qui attendent sur cette condition

```
empty(cv)       /* La file associee est-elle vide? */
```

- Mise en attente d'un processus, donc ajout à la fin de la file:

```
wait(cv);       /* Le processus est mis a la queue. */
```

  $\Rightarrow$ l'accès exclusif au moniteur par le processus se termine

- Réactivation de processus en attente:

```
signal(cv);     /* Reactive le processus en tete. */
```

  Note: aucun effet si la file est vide

Slide 6

5.1.3 Disciplines de signalisation

Deux approches sont possibles pour définir l'effet de `signal`, en tenant compte du fait que le processus qui exécute cette opération est celui qui a alors le contrôle *exclusif* du moniteur:

1. Signaler et Continuer: Le processus qui exécute l'instruction `signal` continue son exécution, donc conserve l'accès exclusif au moniteur. Le processus ayant été signalé sera exécuté plus tard.
2. Signaler et Attendre: le processus qui signale attend pendant que celui qui vient d'être signalé acquiert l'accès exclusif au moniteur.

La première discipline est une approche non-préemptive. Historiquement, elle est plus récente, et c'est maintenant celle la plus couramment utilisée (Unix, Java, Pthreads)

Slide 7

Exemple illustrant les deux stratégies:

```
monitor Semaphore {
  int s = 0;
  cond pos;

  procedure Psem() {
    while (s == 0) wait(pos);
    s = s - 1;
  }

  procedure Vsem() {
     s = s + 1;
     signal(pos);
  }
}
```

Slide 8

Cette mise en oeuvre d'un sémaphore avec un moniteur fonctionne correctement *pour les deux disciplines*. Toutefois, il peut y avoir certaines différences dans l'ordre dans lequel les processus s'exécutent lorsque l'opération `Vsem` est effectuée:

- Signaler et Attendre: le processus réactivé s'exécute immédiatement et décrémente `s`
- Signaler et Continuer: le processus réactivé s'exécute plus tard. Or, il est possible qu'entre temps, un autre processus ait exécuté une opération `Psem`, d'où la nécessité de tester à nouveau la valeur de `s` dans `Psem`

Donc, avec une approche Signaler et Attendre, l'instruction `while` pourrait être remplacée par l'instruction suivante, *mais pas* avec une approche Signaler et Continuer:

```
if (s == 0) wait(pos);
```

Solution alternative qui fonctionne correctement pour les deux disciplines, sans utiliser de `while`: Figure 5.3 (p. 211)

Slide 9

Similitudes/différences entre P/wait et V/signal

- Les opérations wait et P peuvent toutes deux avoir pour effet de suspendre un processus qui exécute cette opération: wait suspend *toujours* le processus, alors que P ne le fait que si la valeur du sémaphore est 0
- signal et V peuvent réactiver un processus suspendu. Par contre, signal n'a aucun effet si aucun processus n'est suspendu, alors que V aura pour effet d'incrémenter la valeur du sémaphore si aucun processus n'est suspendu

Slide 10

5.2 Techniques de synchronisation

5.2.1 Tampons bornés: synchronisation conditionnelle de base

Un moniteur pour un tampon borné définit deux opérations permettant de manipuler ce tampon:

1. deposit: ajoute un élément. Ne suspend que si le tampon est plein.
2. fetch: retire un élément. Ne suspend que si le tampon est vide.

La manipulation de la ressource partagée (le tampon) est faite automatiquement de façon exclusive: *une seule* des deux procédures peut être active à un instant donné.

Exemple: Figure 5.4 (p. 214)

Note: On remarque l'utilisation d'une boucle while, au cas où il y aurait plusieurs producteurs ou consommateurs (discipline Signaler et Continuer).

Slide 11

5.2.2 Processus multiples de lecture et écriture: Signal de diffusion (*broadcast*)

La BD commune aux différents processus *ne peut pas* être encapsulée dans un moniteur, parce que cela ne permettrait pas la présence de lecteurs multiples (moniteur $\Rightarrow$ un seul accès à la fois)

Approche alternative avec moniteur = Utilisation d'un moniteur arbitre (*arbitration monitor*), qui gère les requêtes d'accès à la BD

Quatres requêtes (donc quatre opérations) sont possibles:

1. `request_read`
2. `release_read`
3. `request_write`
4. `release_write`

Slide 12

Invariant devant être satisfait par le moniteur:

```
(nr == 0 || nw == 0) && nw <= 1
```

Avec:

- `nr` = Nombre de lectures en cours
- `nw` = Nombre d'écritures en cours

Exemple: Figure 5.5 (p. 216)

L'opération `relase_write` réactive un seul processus écrivain *et* tous les processus lecteurs:

```
signal(oktowrite);
signal_all(oktoread);
```

Si un processus veut écrire *et* plusieurs processus veulent lire $\Rightarrow$ c'est la politique d'ordonnancement des processus du système qui déterminera quel processus pourra obtenir l'accès à la BD