4 Sémaphores

Motivation = une approche avec *busy waiting* n'est pas intéressante dans un programme avec fils d'exécution multiples, puisque le nombre de processus est généralement plus grand que le nombre de processeurs, donc on ne veut pas faire travailler inutilement un processeur

Approche alternative = utilisation de sémaphores

Slide 1

Sémaphore (selon Larousse) = *Chemin de fer* Signal muni de bras indiquant par leur position si la ligne est libre ou occupée $\Rightarrow$ Permet d'éviter des collisions en assurant l'accès *exclusif* à une section de voie ferrée.

4.1 Syntaxe et sémantique

Sémaphore = sorte *spéciale* de variable contenant un nombre entier *non-négatif* et manipulée uniquement par deux opérations atomiques: P et V

1. P = *Passeren* $\approx$ passer, prendre $\Rightarrow$ Décrémente la variable (à moins qu'elle ne soit déjà 0)

   Utilisée (lorsque déjà 0) pour retarder (suspendre) un processus jusqu'à ce qu'un événement survienne

2. V = *Vrijgeven* $\approx$ relâcher $\Rightarrow$ Incrémente la variable

   Utilisée pour *signaler* un événement et, possiblement, réactiver un processus en attente

Déclarations de sémaphore (notation d'Andrews):

Slide 2

```
sem s1;          /* sem s1 = 0; */

sem s2 = 1;

sem forks[5] = ([5] 1);
```

Après initialisation, les seules opérations permises sont P et V:

```
P(s):  < await(s > 0) s = s - 1; >

V(s):  < s = s + 1; >
```

Slide 3

Sémaphore général vs. sémaphore binaire:

- Sémaphore général = peut prendre n'importe quelle valeur non-négative
- Sémaphore binaire = la valeur peut être uniquement 0 ou 1

Les sémaphores peuvent être utilisés tant pour la réalisation des sections critiques que pour diverses formes de synchronisation conditionnelle

Remarque: la plupart des mise en oeuvre des sémaphores assurent que les processus en attente sont réactivés *dans l'ordre* dans lequel ils ont été suspendus sur le sémaphore $\Rightarrow$ Équité dans l'ordonnancement des processus

Slide 4

4.2 Problèmes et techniques de base

4.2.1 Sections critiques: Exclusion mutuelle

Correpondance entre cadenas et sémaphore:

- Acquérir le cadenas $\approx$ obtenir le sémaphore (P)
- Libérer le cadenas $\approx$ libérer le sémaphore (V)

```
sem mutex = 1;

process CS[i = 1 to n] {
  P(mutex);
  section critique;
  V(mutex);
  section non-critique;
}
```

Différence majeure entre cadenas et sémaphore: P ne requiert pas que le processus attende de façon active (pas de *busy waiting*)

Slide 5

4.2.2 Barrières: Signaler des événements

Sémaphore *de signalisation* = sémaphore binaire dont la valeur initiale est 0

Version simplifiée du problème = barrière uniquement pour 2 processus
(voir transparent suivant):

1. Le processus signale son arrivée
2. Le processus attend ensuite que l'autre ait aussi signalé son arrivée

Solution pour n processus: généralisation, par ex., à l'aide d'arbres de synchronisation, comme dans les sections 3.4.2 et 3.4.3

Autre approche possible pour n processus: solution avec processus coordonnateur

Slide 6

```
sem arrive1 = 0; arrive2 = 0;

process Worker1 {
  ...
  V(arrive1);        /* Signaler mon arrivee. */
  P(arrive2);        /* Attendre l'arrivee de l'autre. */
  ...
}

process Worker2 {
  ...
  V(arrive2);        /* Signaler mon arrivee. */
  P(arrive1);        /* Attendre l'arrivee de l'autre. */
  ...
}
```

Slide 7

4.2.3 Producteur et consommateur: Sémaphores binaires inversés (*split binary semaphores*)

L'alternance entre le producteur et le consommateur peut être obtenue, comme on l'a vu dans les versions Threaded-C, en faisant en sorte que les processus producteur et consommateur se signalent l'un et l'autre lorsque le tampon devient vide/plein

Solution avec sémaphores $\Rightarrow$ Utilisation de deux sémaphores binaires inversés (transparent suivant)

On a alors les conditions suivantes:

`empty = 1 <=> full = 0`

`empty = 0 <=> full = 1`

Donc, `empty = 1` et `full = 1` est impossible, ce qui assure l'exclusion mutuelle

Slide 8

```
typeT buf;
sem empty = 1; full = 0;

process Producer {
  while(true) {
    ... produire quelque chose dans data ...
    P(empty);        /* On attend que buf soit libre. */
    buf = data;
    V(full);         /* On signale que buf est plein. */
  }
}

process Consumer {
  while(true) {
    P(full);         /* On attend que buf soit plein. */
    result = buf;
    V(empty);        /* On signale que buf est libre. */
    ... traiter result ...
  }
}
```

4.2.4 Tampons bornés: Déterminer les ressources disponibles

Exemple précédent ⇒ *alternance stricte* entre producteur et consommateur: le tampon (`buf`) ne peut contenir qu'un seul élément

Acceptable seulement si les données sont produites/consommées à des vitesses similaires et les coûts d'attente/réactivation ne sont pas élevés

Une solution plus intéressante serait d'utiliser un tampon pouvant contenir *plusieurs* éléments, permettant ainsi une plus grande flexibilité au niveau de la vitesse des processus

Slide 9

Stratégie = utilisation d'un tampon *circulaire* de taille bornée `n`:

- Dépot d'un item par le producteur à la "queue" du tampon:

```
buf[rear] = data;  rear = (rear+1) % n;
```

- Retrait d'un item par le consommateur à la "tête" du tampon:

```
result = buf[front];  front = (front+1) % n;
```

Effets =

- Le producteur peut produire et déposer un item tant qu'il y a un espace libre dans le tampon
- Le consommateur peut retirer un item tant qu'il y en a un de disponible
- Les deux peuvent modifier en même temps le tampon, en autant que cela soit sur des parties différentes

Solution: Voir transparent suivant

Slide 10

Rôle des sémaphores (*counting semaphores*)

- `empty` = nombre de positions libres dans le tampon
- `full` = nombre de positions occupées dans le tampon

Autre version d'un problème similaire = Plusieurs producteurs et plusieurs consommateurs: Figure 4.5 (p. 163)

Slide 11

```
typeT buf[n];
int front = 0, rear = 0;
sem empty = n; full = 0;

process Producer {
  while(true) {
    ... produire quelque chose dans data ...
    P(empty);
    buf[rear] = data; rear = (rear+1) % n;
    V(full);
  }
}

process Consumer {
  while(true) {
    P(full);
    result = buf[front]; front = (front+1) % n;
    V(empty);
    ... traiter result ...
  }
}
```

Slide 12

4.4 Processus multiples de lecture et d'écriture (*Readers and writers*)

Autre problème classique: Deux sortes de processus — des processus qui lisent, d'autres qui écrivent — partagent une base de données.

Chaque opération sur la BD doit donner lieu à une transaction qui examine ou modifie la BD.

Pour prévenir les interférences entre les transactions, celles qui *modifient* la BD doivent être exécutées de façon atomique et exclusive. Par contre, si aucune transaction d'écriture n'est en cours, plusieurs transactions de lecture peuvent avoir lieu en parallèle.

Problème d'exclusion mutuelle *sélective*: un processus qui lit la BD doit exclure les processus qui veulent y écrire, mais sans exclure les autres processus qui veulent lire

Slide 13

4.4.1 Processus multiples de lecture et d'ecriture comme problème d'exclusion mutuelle

Solutions simples:

- Figure 4.9 (p. 169): Avec un sémaphore `rw` pour assurer l'exclusion entre processus de lecture vs. écriture et une section atomique dans les processus lecteurs (`<...>`) pour manipuler la variable `nr`, laquelle indique le nombre de lecteurs en train de s'exécuter
- Figure 4.10 (p. 170): Comme la version précédente, mais où la section `<...>` est remplacée par la manipulation d'un autre sémaphore (`mutexR`)

Remarque: Ces deux solutions donnent la préférence aux lecteurs $\Rightarrow$ si un lecteur est actif et que deux processus veulent accéder la BD, un en lecture, l'autre en écriture, alors celui voulant effectuer la lecture sera favorisé

Slide 14

4.6 Étude de cas: *Pthreads*

Les mécanismes de création de processus de style Unix (`fork`, `wait`, `pipe`) sont très coûteux et ne peuvent être utilisés que pour des programmes avec un nombre restreint de processus (processus poids lourd)

De plus, ces mécanismes varient d'un système d'exploitation à un autre (Unix vs. MVS vs. ...)

Au milieu des années 90, diverses *librairies* de *thread*s (= processus poids léger) furent introduites (souvent associées au langage C)

Une des plus connues = POSIX = *Portable Operating System Interface*, d'où le nom *Pthreads*

Pthreads définit des douzaines de fonctions, dans le cadre du langage C, pour la programmation concurrente avec *thread*s

Slide 15

4.6.1 Création de *thread*s

- Inclusion de la libraire:

```
#include <pthread.h>
```

- Déclaration de variables décrivant les attributs du *thread* ainsi que son identification:

```
pthread_attr_t tattr;
        /* Attributs du thread. */
pthread_t      tid;
        /* Descripteur (identifiant) du thread. */
```

- Initialisation des attributs:

```
pthread_attr_init(&tattr);
        /* Utilisation des attributs par defaut. */
pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
        /* Execution du thread en fonction
           de l'ensemble des threads actifs. */
```

Slide 16

- Création du *thread*:

```
pthread_create(&tid, &tattr, func, arg);
```

≈ `TOKEN` de Threaded-C: fonction (`func`) + argument

L'opération retourne 0 si le *thread* a été créé correctement; dans ce cas, `tid` contient l'identifiant (le descripteur) du *thread* créé

- Attente pour qu'un processus se termine: Un parent (créateur d'un *thread*) peut attendre qu'un enfant se termine:

```
pthread_join(tid, &result);
```

La valeur dans `result` (sauf si on a indiqué `NULL`) est celle spécifiée par le *thread* lorsqu'il se termine avec:

```
pthread_exit(return_value);
```

Note: Si la fonction pour un *thread* se termine sans un appel explicite à `pthread_exit`, alors `NULL` est retournée

Slide 17

4.6.2 Sémaphores

Les *thread* communiquent entre eux par l'intermédiaire de variables déclarées de façon globale (déclarées à l'extérieur des fonctions associées à chacun des *thread*s)

Les *thread*s peuvent se synchroniser entre eux par l'intermédiaire de sémaphores, cadenas ou moniteurs (décrits au chap. 5)

Utilisation de sémaphores:

- Inclusion du fichier approprié

```
#include <semaphore.h>
```

- Déclaration d'une variable globale de type `sem_t`:

```
sem_t mutex;
```

Slide 18

- Initialisation

```
sem_init(&mutex, SHARED, 1);
```

  L'attribut SHARED indique si le sémaphore peut être partagée entre les *thread*s d'un même processus (0) ou entre les *thread*s de processus indépendants (1)

- Les opérations de base:

```
sem_wait(&mutex);   /* P(mutex); */
sem_post(&mutex);   /* V(mutex); */
```

De nombreuses autres opérations sont disponibles: attente conditionnelle, examen de la valeur courante du sémaphore, destruction d'un sémaphore

4.6.3 Exemple: Une version simple du producteur et du consommateur

Voir Figure 4.15 (p. 187)