Problèmes classiques de synchronisation des processus

Achraf Othman

Revue

- Qu’est-ce qu’un sémaphore ?
- Qu’est-ce qu’un moniteur?
- Pourquoi est-ce que les moniteurs sont utiles?

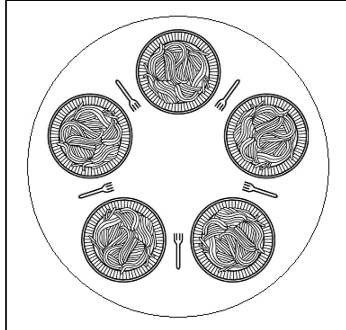
Synopsis

- Nous allons voir trois problèmes intéressants qui sont reliés à la communication interprocessus
  - Le dîner des philosophes
    - Modélise l'accès aux E/S
  - Le problème des lecteurs et rédacteurs
    - Modélise l'accès à une base de données
  - Le problème du barbier endormi
    - Modélise les systèmes de queues

Le problème du dîner des philosophes

- L'abstraction d'un philosophe: Il alterne entre des périodes où il mange et où il pense
- Cinq philosophes sont assis entour d'une table avec des assiettes de spaghetti très glissant
  - Chacun a une assiette et une fourchette entre chaque assiette
  - Quand un philosophe a faim, il essaie de prendre une fourchette de chaque côté (une à la fois, dans n'importe quel ordre)
  - Si il réussi, le philosophe mange pour un certain temps et dépose les deux fourchettes
- Comment est-ce qu'un programme peut être dessiné pour assurer que les philosophes ne s'embourbent pas?

Le problème du dîner des philosophes



```
#define N 5
void philosopher(int i){
      while (TRUE) {
            think( );
            take_fork(i);
            take_fork((i+1) % N);
            eat( );
            put_fork(i);
            put_fork((i+1) % N);
      }
}
```

- `take_fork()` est un appel qui bloque.
- Quel est le problème avec cette solution?

Le problème du dîner des philosophes

- Corrections possibles:
  - Un philosophe prend une fourchette (gauche) en premier. Si l'autre n'est pas disponible, il dépose la fourchette qu'il a pris et attend pour un certain temps (constant) avant de réessayer
    - Problème: Si tous les algorithmes commencent simultanément il est possible que tous les philosophes vont prendre la fourchette de gauche en même temps, voir que la fourchette de droite manque, déposer la fourchette de gauche et répéter infiniment
    - Notez: quand les processus sont dans un état qui n'est pas de l'interblocage mais qu'ils ne font pas de progrès cela s'appel une privation de ressources (**starvation ou livelock**)

Le problème du dîner des philosophes

- Corrections possibles:
  - ➢Pour prévenir la privation, faire attendre les philosophes un montant de temps aléatoire avant de réessayer
    - Problème: Un vrai temps aléatoire pourrait causer le même problème à arriver encore, par contre pas vraisemblable...
    - Une bonne solution pour Ethernet, mais que ce passe-t'il si les ressources sont requises pour un réacteur nucléaire? Quelques millisecondes nous séparent d'une catastrophe...

Le problème du dîner des philosophes

- Corrections possibles:
  - ➢Utiliser un sémaphore pour protéger les cinq instructions après l'appel `think()`
    - Problème: Pas efficace. Interblocage n'arrive plus, mais seulement un philosophe peut manger à la fois. Pas très bon pour la multiprogrammation

```
#define N 5
void philosopher(int i){
        while (TRUE) {
                think( );
                take_fork(i);
                take_fork((i+1) % N);
                eat( );
                put_fork(i);
                put_fork((i+1) % N);
        }
}
```

```
#define LEFT  (i+N-1)%N
#define RIGHT (i+1)%N

void philosopher(int i) {
  while(TRUE) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}

void take_forks(int i) {
  down(&mutex);
  state[i] = HUNGRY;
  test(i);
  up(&mutex);
  down(&s[i]);
}

void put_forks(i) {
  down(&mutex);
  state[i] = THINKING
  test(LEFT);
  test(RIGHT);
  up(&mutex);
}

void test(i) {
  if(state[i] == HUNGRY) && state[LEFT]!=EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&s[i]);
  }
}
```

Le problème des lecteurs et rédacteurs

- Plusieurs processus qui essaient d'écrire et lire à/sur une base de données
- On peut avoir autant de processus que l'on veut qui lisent de la base de données simultanément
- Seulement un seul processus peut accéder la base de données si il écrit

Le problème des lecteurs et rédacteurs

- Difficultés:
  - ➢Une exclusion mutuelle simple ne marche pas parce que nous voulons plusieurs lecteurs dans la base de données en même temps.
    - On ne peut pas utiliser un sémaphore binaire sur la base de données
  - ➢Un sémaphore qui compte simplement nous rend la tâche difficile pour empêcher un rédacteur d'entrer pendant les lectures

Le problème des lecteurs et rédacteurs

- Solution:

```
typedef int semaphore;
semaphore mutex=1;
semaphore db=1;
int rc=0;
```

```
void writer(void) {
  while(1) {
    think_up_data();
    down(&db);
    write_data_base();
    up(&db);
  }
}
```

```
void reader(void) {
  while(1) {
    down(&mutex);
    rc = rc+1;
    if(rc==1)
      down(&db);
    up(&mutex);
    read_dat_base();
    down(&mutex);
    rc = rc-1;
    if(rc==0)
      up(&db);
    up(&mutex);
    use_data_read();
  }
}
```

Le problème des lecteurs et rédacteurs

- Deux choses sont intéressantes avec cette solution:
  - Un des lecteurs successifs fait `down()` seulement une fois sur le sémaphore binaire `db`, le premier lecteur le fais
    - Ceci permet au lecteurs multiples d'entrer dans la base de données simultanément
  - Parce que `down()` sur le sémaphore `db` est protégé en dedans du `mutex`, les lecteurs qui essaient d'entrer dans la base de données après le premier vont être bloqués sur le `mutex`, pas sur le `db`

Le problème du barbier qui dort

- Le barbier est dans son salon. Si il n'a pas de client, il s'endort dans sa chaise
- ***n*** chaises pour que les clients attendent
- Le premier client qui arrive doit éveiller le barbier
- Les clients subséquents s'assoient si il y a des chaises, sinon il quittent le salon
- Difficile de développer une solution sans concurrence critique

```
#define CHAIRS 5                    /* # chairs for waiting customers */
typedef int semaphore;              /* use your imagination */
semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void) {
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}

void customer(void) {
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Le problème du barbier qui dort

- Notes à propos de la solution:
  - La variable `waiting` est une copie du sémaphore `customers`. Pourquoi?
    - Parce que le sémaphore `customers` n'est pas protégé en dedans du `mutex` dans le processus du barbier, il ne peut pas être lu en sécurité
  - On a pas besoin d'avoir une boucle dans le design de `customer` parce que chaque `customer` a besoin d'une coupe de cheveux seulement une fois
    - Par contre le barbier doit avoir une boucle
  - Le design initial de ce genre de système peut vous faire saigner au cerveau

Quiz time

Questions?

TOUR OF COMP ENG

OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.

NINE NINE NINE NINE NINE NINE

ARE YOU SURE THAT'S RANDOM?

THAT'S THE PROBLEM WITH RANDOMNESS. YOU CAN NEVER BE SURE.