Communications et synchronisations

Le problème des philosophes (Dijkstra 1965)

— Cinq philosophes sont assis autour d'une table ronde

— Chaque philosophe a devant lui un plat de spaghettis tellement glissants qu'il faut deux fourchettes pour pouvoir les manger

— Une fourchette sépare chaque plat (il y a donc autant de plats que de philosophes que de fourchettes)

— Le comportement de chaque philosophe est le suivant :

```
while (1) {
  penser();
  obtenir_fourchettes();
  manger();
  relacher_fourchettes();
}
```

— Les fourchettes représentent des ressources que le processus doit détenir exclusivement pour pouvoir continuer à travailler

Le problème des philosophes

— Écrire le code des fonctions `obtenir_fourchettes` et `relacher_fourchettes`, de telle sorte que les contraintes suivantes soit satisfaites :
  — Un seul philosophe à la fois peut détenir une fourchette
  — Les processus ne doivent pas arriver à un interblocage
  — Plusieurs philosophes doivent pouvoir manger en même temps
  — Un philosophe ne doit mourir de faim en attendant une fourchette
— On donne le code suivant :

```
#define NbPhilo 5
int gauche(int i) { return (i+1)%NbPhilo; }
int droite(int i) { return i; }
pthread_mutex_t fourchettes[NbPhilo];
```

— les mutex sont initialisés

```
for (i=0; i < NbPhilo; i++) { pthread_mutex_init(&fourchettes[i], NULL); }
```

Le problème des philosophes : solution 1 (philo-1.c)

```
pthread_mutex_t fourchettes[NbPhilo];

void obtenir_fourchettes(int i) {
  pthread_mutex_lock(&fourchettes[gauche(i)]);
  pthread_mutex_lock(&fourchettes[droite(i)]);
}
void relacher_fourchettes(int i) {
  pthread_mutex_unlock(&fourchettes[gauche(i)]);
  pthread_mutex_unlock(&fourchettes[droite(i)]);
}
```

— Interblocage

Le problème des philosophes : solution 2 (philo-2.c)

```
int fourchettes[NbPhilo];
pthread_mutex_t mutex;
void obtenir_fourchettes(int i) {
  int ok = 0;
  while (!ok) {
    pthread_mutex_lock(&mutex);
    if ( fourchettes[gauche(i)] && fourchettes[droite(i)] ) {
      fourchettes[gauche(i)] = fourchettes[droite(i)] = 0;
      ok = 1;
    }
    pthread_mutex_unlock(&mutex);
  }
}
void relacher_fourchettes(int i) {
  pthread_mutex_lock(&mutex);
  fourchettes[gauche(i)] = fourchettes[droite(i)] = 1;
  pthread_mutex_unlock(&mutex);
}
```

— Famine

Le problème des philosophes : solution 3 (philo-3.c)

```
pthread_mutex_t fourchettes[NbPhilo];

void obtenir_fourchettes(int i) {
  int f1, f2;

  if (i == 0) {
    f1 = droite(i);
    f2 = gauche(i);
  } else {
    f1 = gauche(i);
    f2 = droite(i);
  }
  pthread_mutex_lock(&fourchettes[f1]);
  pthread_mutex_lock(&fourchettes[f2]);
}
void relacher_fourchettes(int i) {
  pthread_mutex_unlock(&fourchettes[gauche(i)]);
  pthread_mutex_unlock(&fourchettes[droite(i)]);
}
```

— OK

Les sémaphores

- Sémaphores
  - Un compteur
  - Une file d'activités en attente
- Compteur associé au sémaphore
  - nombre de jetons
  - valeur positive = nombre d'activités pouvant acquérir librement la ressource
  - valeur négative = nombre d'activités bloquées en attente de la ressource

Les sémaphores

- Primitives pour l'utilisation des sémaphores
  - Création / initialisation : nombre initial de jetons
  - Prendre : un processus exécute P.
    - classiquement noté P (puis-je ?)
    - attribue un jeton au processus demandeur s'il en reste un (le nombre de jetons est différent de zéro)
    - bloque le processus sinon en l'ajoutant à la file du sémaphore.
  - Libérer : le processus exécute V.
    - classiquement noté V (vas-y !)
    - rend un jeton
    - débloque un processus en attente dans la file, s'il en exite un.

Syntaxe des sémaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

`sem_init(sem, p, v)` :

- `v` : valeur d'initialisation du sémaphore ;
- `p` = 0 : le sémaphore est partage entre les proc. légers du processus et doit être visible par tous les processus légers.
- Sinon : le sémaphore est partagé par tous les processus.

`sem_wait(sem)` : décrémente le sémaphore pointé par `sem`. Si sa valeur est supérieure à 0, la fonction termine immédiatement. Si la valeur est nulle, l'appel est bloquant jusqu'à ce que la décrémentation soit possible ou que l'appel soit interrompu un signal.

`sem_post(sem)` : incrémente le sémaphore pointé par `sem`. Si cette valeur devient strictement positive alors un autre processus en attente (exécutant un `sem_wait`) est debloqué.

Philosophes avec sémaphores (phio-sem.c)

Un sémaphore global + un tableau de mutex pour faire en sorte que deux philosophes voisins ne prennent pas la même fourchette en même temps.

Une synchronisation locale seulement ; plus efficace que les solutions précédentes.

```
pthread_mutex_t fourchettes[NbPhilo];
sem_t places;

void obtenir_fourchettes(int i) {
  sem_wait(&places);
  pthread_mutex_lock(&fourchettes[gauche(i)]);
  pthread_mutex_lock(&fourchettes[droite(i)]);
}
void relacher_fourchettes(int i) {
  pthread_mutex_unlock(&fourchettes[gauche(i)]);
  pthread_mutex_unlock(&fourchettes[droite(i)]);
  sem_post(&places);
}

int main() {
  ...
  sem_init(&places, 0, NbPhilo-1);
  ...
}
```

Exclusion mutuelle avec les sémaphores (mutex-sem.c)

```
struct mutex {
  sem_t mutex_sem;
};

void mutex_init(struct mutex *m) {
  sem_init(m.mutex_sem, 0, 1);
}
void mutex_lock(struct mutex *m) {
  sem_wait(m.mutex_sem);
}
void mutex_unlock(struct mutex *m) {
  sem_post(m.mutex_sem);
}
```

Problème du producteur consommateur

— Deux activités
  — un producteur et un consommateur (ou plusieurs producteurs et plusieurs consommateurs)
  — relié par un tampon de taille bornée
  — producteur ne peut pas produire si le tampon est plein
  — consommateur ne peut pas consommer si le tampon est vide
  — producteur et consommateur ne doivent pas travailler sur le même élément
— Problème classique
  — pipe et fifo
  — queues de messages

prod-cons.c

```
sem_t mutex;
sem_t vide;  // nb. emplacements libres
sem_t plein; // nb. emplacements utilises

pthread_t prod;
pthread_t cons;

int tampon[N];
int libre = 0;    // index du premier emplacement libre
int prochain = 0; // index de la prochaine valeur \‘a lire
```

prod-cons.c

```
void *producteur(void *arg) {
  int objet;  // un objet de type arbitraire (ici entier)
  while (VRAI) {
    objet = 1 + (int)(100.0*rand()/RAND_MAX+1.0);
    sem_wait(&vide); /* on decremente le nombre de places vides */
    sem_wait(&mutex);
    tampon[libre] = objet;
    printf("Prod: tampon[%d] = %d\n", libre, objet);
    libre = (libre + 1) % N;
    sem_post(&mutex);
    sem_post(&plein);
  }
  return NULL;
}
```

Le sémaphore plein permet d'assurer que l'on ne lit pas dans un buffer vide.

prod-cons.c

```
void *consommateur(void *arg) {
  int objet;
  while (VRAI) {
    sem_wait(&plein);
    sem_wait(&mutex);
    objet = tampon[prochain];
    printf("Cons: tampon[%d] = %d\n", prochain, objet);
    prochain = (prochain + 1) % N;
    sem_post(&mutex);
    sem_post(&vide);
    sleep(2);
  }
}
```

prod-cons.c

```
int main(int argc, char **argv) {
  sem_init(&mutex, 0, 1);
  sem_init(&vide, 0, N); /* on initialise le semaphore a N */
  sem_init(&plein, 0, 0); /* force a commencer par executer un sem_post */

  pthread_create(&prod, NULL, producteur, NULL);
  pthread_create(&cons, NULL, consommateur, NULL);

  pthread_join(prod, NULL);
  pthread_join(cons, NULL);
  return 0;
}
```

Les moniteurs

— Structure de programmation
  — un module de programme
  — contrôle les accès à des données partagées
— Variables d'état
  — partagées entre les procédures du module
  — encapsulées dans le module
— Synchronisation entre les activités concurrentes qui appellent les procédures

Mécanisme d'exécution du moniteur

- Exclusion mutuelle entre les entrées du moniteur
  - à un instant donné : une seule activité dans le moniteur
  - donc protection des variables d'état
- Si une seconde activité cherche à entrer dans le moniteur
  - bloquée jusqu'à la libération du moniteur par l'activité précédente
  - file d'attente des activités bloquées en entrée du moniteur

Moniteur en Java (Counter.java)

En Java, le mot clef `synchronized` associé à une déclaration de méthode assure que l'appel à la méthode est exécuté en exclusion mutuelle, c-a-d, exécuté atomiquement.

```
public class Counter {
    private int valeur = 0;

    public synchronized void incr() {
        valeur += 1;
    }

    public synchronized int get() {
        return valeur;
    }
}
```

Moniteur en Java (Exemple.java)

```
public class Exemple  extends Thread {
    private static Counter cpt = new Counter();
    public static void main (String[] args) {
        Exemple t1 = new Exemple("T1");
        Exemple t2 = new Exemple("T2");
        t1.start();
        t2.start();
    }
    public Exemple(String nom) { super(nom); }
    public void run() {
        for (int i=0; i <1000000; i++) {
            cpt.incr();
        }
        System.out.println(getName() + " : " + cpt.get());
    } }
```

Moniteur en Java (Mutex.java)

```
public class Mutex {
    private boolean jeton = true;
    public synchronized void unlock() {
        jeton = true;
        notifyAll();
        return ;
    }
    public synchronized void lock() {
        while (!jeton) {
            try { wait(); }
            catch(InterruptedException e) {}
        }
        jeton = false;
        notifyAll();
        return ;
    } }
```