.of. Yann Thoma

heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Programmation Concurrente (PCO)
Semestre printemps 2013-2014
Contrôle continu 2
13.06.2014

Prénom: Antoine

Nom: Messerli

- Aucune documentation n'est permise, y compris la feuille de vos voisins
- La calculatrice n'est pas autorisée
- Aucune réclamation ne sera acceptée en cas d'utilisation du crayon
- Ne pas utiliser de couleur rouge

Question	Points	Score
1	15	15
2	17	12
3	18	12
Total:	50	39

Note: 4,9

Question 1: 15 points

Nous sommes intéressés par simuler le comportements des utilisateurs du réseau de vélos Publibike. Pour ce faire il vous est demandé d'implémenter la classe `Site`, qui gère l'accès aux vélos d'un site. Le site est initialisé avec le nombre de bornes dont il dispose. La classe exporte en outre deux méthodes permettant de déposer un vélo et d'en prendre un. Ces deux méthodes sont évidemment bloquantes. Il n'est en effet possible de prendre un vélo que s'il y en a un disponible, et il n'est possible de déposer un vélo que s'il y a une borne de libre.
Complétez la déclaration de la classe et implémentez les 4 méthodes en utilisant un moniteur POSIX.

Rappel sur les moniteurs Posix :
- `pthread_mutex_init(&m,0)` : initialise un mutex ;
- `pthread_mutex_destroy(&m)` : détruit un mutex ;
- `pthread_lock(&m)` : verrouille un mutex ;
- `pthread_unlock(&m)` : déverrouille un mutex ;
- `pthread_cond_init(&c,0)` : initialise une variable condition ;
- `pthread_cond_destroy(&c)` : détruit une variable condition ;
- `pthread_cond_wait(&c,&m)` : se met en attente sur une variable condition ;
- `pthread_cond_signal(&c)` : signale une condition ;

```
class Site {
    // Déclaration de variables
  unsigned int nbBornes
  unsigned int nbVelos
   pthread_t mutex;
   pthread_cond c;
public:
  Site(unsigned int nbBornes);
  ~Site();
  void prendVelo();
  void deposeVelo();
};
```

```
Site(unsigned int nbBornes){
this.nbBornes = nbBornes;        nbVelos = 0;
 init(&mutex,0),
 int(&c, 0);
}

~Site(){
    mutex_destroy(&mutex);
    cond_destroy(&c);
}
```

```
void prendVelo() {
    lock(&mutex);

    while (nbVelos == 0) { // Attend un velo
        wait(&c, &mutex);
    }
    nbVelos--;
    signal(&c);
    unlock(&mutex);
}

void deposeVelo() {
    lock(&mutex);
    while (nbVelos == nbBornes) { // Attend qu'une borne se libère
        wait(&c, &mutex);
    }
    nbVelos++;
    signal(&c);
    unlock(&mutex);
}
```

Dangereux avec une seule variable condition

Ok une seule variable condition n'est pas bon car la solution ne fonctionne que si la file d'attente est gérée selon un mode FIFO

Prod/Cons

Question 2: 17 points

Dans le cadre du Mondial de foot nous devons gérer un bar au CULTES (Centre Urbain de Lutte Tenace à l'Encontre des Supporters). Le bar est géré de la manière suivante : Il dispose d'une table avec d'un côté des remplisseurs de verres, et de l'autre des distributeurs de boissons aux clients. Les remplisseurs ont pour tâche de déposer des verres de boisson sur la table, en spécifiant le type de boisson (bière [1] ou lait [2]). Les distributeurs attendent les commandes des clients et doivent récupérer la boisson correspondante sur la table. Il est à noter qu'un client peut désirer de la bière, du lait, ou simplement avoir soif et n'avoir aucun désir de type de boisson particulier. Dans ce cas s'il y a quelque chose sur la table le distributeur devra le lui donner.

Proposez une implémentation de la classe `Desk` en implémentant les 4 méthodes et en spécifiant les variables membres. Votre implémentation devra être faite grâce à un moniteur POSIX.

```
class Desk {
public:
  enum BOISSON {BIERE,LAIT,ANY};

  Desk();
  ~Desk();
  void poseBoisson(BOISSON b);
  void prendBoisson(BOISSON b);
};
```

unsigned int biere, lait; sem_t mutex; Sem_cond c;

3 variables conditions
biere
lait
any

Desk() {
biere = 0
lait = 0
sem_init(&mutex, 0);
cond_init(&c, 0);

}

~Desk() {
sem_destroy(&mutex);
cond_destroy(&c);

}

1. Pour les supporters
2. Pour les rockers qui ne boivent que du lait

```
void poseBoisson(BOISSON b) {
    lock(&mutex);
    if(b == BIERE) {
        biere++;
    } else { // lait
        lait++;
    }
    signal(&c); // ça ne réveille pas forcément le bon...
    unlock(&mutex);
}
void PrendBoisson(BOISSON b) {
    lock(&mutex);
    if(b == BIERE) {
        while(biere == 0) {
            lock(&c, &mutex); // attendre sur biere
        }
        biere--;
    } else if(b == LAIT) {
        while(lait == 0) {
            lock(&c, &mutex); // attendre sur lait
        }
        lait--;
    } else if(b == ANY) {
        while(lait == 0 && biere == 0) {
            lock(&c, &mutex); // attendre sur any
        }
        if(biere > 0)
            biere--;
        else
            lait--;
    }
    unlock(&mutex);
}
```

Question 3: 18 points

Nous devons réaliser une barrière de synchronisation un peu spéciale. Il s'agit de combiner des threads "Oxygène" avec des threads "Hydrogène" pour créer de l'"Eau". Pour passer la barrière, un thread doit appeler la fonction `bond(Type)`, et respecter les contraintes suivantes :

1. Lorsqu'un thread "Oxygène" arrive à la barrière et qu'aucun thread "Hydrogène" n'est présent, il doit attendre deux threads "Hydrogène" pour passer la barrière.
2. Lorsqu'un thread "Hydrogène" arrive et qu'aucun autre thread n'est présent, il doit attendre un thread "Oxygène" et un "Hydrogène" pour continuer.

C'est à vous d'en dériver le comportement des threads dans les autres cas de figure afin de rester cohérent avec ces deux conditions.
Implémentez la classe suivante à l'aide de sémaphores POSIX.

Rappel sur les sémaphores Posix :
- `sem_init(&a,0,n)` : correspond à initialiser le sémaphore `a` à `n` (`n` doit être ≥ 0);
- `sem_wait(&a)` : correspond à `P(a)`;
- `sem_post(&a)` : correspond à `V(a)`.

```
class Bonding {
  public:
    enum Type {OXYGEN,HYDROGEN};
    Bonding();
    ~Bonding();
    void bond(Type type);
}
```

unsigned int nbH;
unsigned int nbO;

sem_t mutex, waitOxygen, waitHydrogen, mutex
hydrogen, oxygen, libereH,
libereO;

Bonding() {
  init(&mutex, 0, 1);
  init(&waitOxygen, 0, 0);
  init(&waitHydrogen, 0, 0);
  init(&hydrogen, 0, 0);
  init(&oxygen, 0, 0);
}

init(&libereH, 0, 0);
init(&libereO, 0, 0);

nbH=0;
nbO=0;

~Bonding() {

}

```
void bond(Type type) {

    if(type == OXYGEN) {
        wait(&mutex);
        if(nbH == 0) {  // pas d'hydrogène
            post(&mutex); nbO++;
            wait(&waitHydrogen);
            wait(&waitHydrogen); // Attend 2x de l'hydrogène
            wait(&mutex);
            nbO--; // n'est plus en attente d'hydrogène
            post(&mutex);
            post(&waitOxygen);
            post(&waitOxygen); // libère 2x hydrogen
        } else { // un hydrogène en attente
            post(&mutex);
            post(&Oxygen);
            wait(&libereO);
        }
    } else if(type == HYDROGEN) {

        wait(&mutex);
        if(nbO > 0) {
            post(&mutex);
            post(&waitHydrogen); // libère 1x l'oxygène
            wait(&waitOxygen); // Attend que l'oxygen libéré
        } else if(nbH == 0) { // pas d'hydrogen en attente
            nbH++;
            post(&mutex);
            wait(&hydrogen); // Attend sur 1 hydrogène
            wait(&Oxygen); // Attend sur un oxygène
            wait(&mutex);
            nbH--; // il n'est plus en attente
            post(&mutex);
            post(&libereH); // libère l'hydrogène
            post(&libereO); // libère l'oxygen
        } else {}
```

préemption par un autre oxygène possible

Compliqué !

```
(1) } else { // hydrogene on attend
        sem_post(&mutex);
        sem_post(&hydrogen);
        sem_wait(&liberer);
    }
}
```

mutex = 1
~~Semaphore~~
with = 0
release = 0

~~des type~~

```
case oxygen :
    sem_wait(mutex);
    sem_wait(with);
    sem_wait(with);

    sem_post(release);
    sem_post(release);

    sem_post(mutex);
```

case hydrogen

```
    ~~sem_wait(~~
    sem_post(with);
    sem_wait(release);
    ~~sem_post(~~
```