Degouey Corentin

IDU3 G2

Lien du code : https://github.com/skiiing73/Synchronisation_processus_INFO632

TP3/TP4 INFO632

TP3 :

Question1 - signalisation:

Le but de cette première question est d'introduire la notion de sémaphore. Pour cela nous voulons appliquer un ordre de priorité sur deux taches a et b afin que la tache b ne s'exécute qu'après la tâche a.

Pour cela nous initialisons la sémaphore a 0 et nous définissons les threads comme ceci :

```c
void *p1(void *arg) {
    a();
    sem_post(sync_x);
    return NULL;
}

void *p2(void *arg) {
    /* wait for the first thread */
    sem_wait(sync_x);
    b();
    return NULL;
}
```

Ici nous n'incrémentons donc la sémaphore qu'après que la tache a soit faite.

La tache b ne peut se lancer tant que la sémaphore n'a pas été incrémentée

Le résultat est le suivant :

```
corentin@rogstrix15:~/TP3$ ./ex1
X = 0
X = 55
```

La tâche a s'effectue bien avant la tâche b.

Question2 - rendez-vous:

Process One

```
a1;
wait(&sem2);
signal(&sem1);
a2;
```

Process Two

```
b1;
wait(&sem1);
signal(&sem2);
b2;
```

La solution ci-dessus ne marchera pas si les deux sémaphores sont initialisées a 0. En effet après avoir effectué a1, le thread1 va attendre que sem2 >0 ce qui ne pourra arriver car le thread2 attend que sem1>0. Il va donc y avoir un blocage. Pour cela il faut inverser le wait et le signal pour chaque thread. On aura donc :

```c
void *p1(void *arg) {
    a1();
    sem_post(sem2);//permet de liberer sem2 quand a1 a fini
    sem_wait(sem1);//attend que b1 ait fini
    a2();
    return NULL;
}

void *p2(void *arg) {
    b1();
    sem_post(sem1);//permet de liberer sem1 quand b1 a fini
    sem_wait(sem2);//attend que a1 ait fini
    b2();
    return NULL;
}
```

Question3 - exclusion mutuelle:

```
init(&mutex, 1);

Thread A
wait(&mutex);
/* CS */
count = count + 1;
signal(&mutex);

Thread B
wait(&mutex);
/* CS */
count = count + 1;
signal(&mutex);
```

Le pseudo-code ci-dessus permet de garantie l'exclusion mutuelle d'une variable. En effet lorsqu'un des deux threads est lancé la sémaphore (initialement à 1) passe à 0. La sémaphore étant incrémenté uniquement à la fin du thread il est impossible pour le thread2 de se lancer.

Afin de pouvoir créer 10 threads qui se partage un valeur x le thread est le suivant :

```c
/* Thread function */
void *thread(int i)
{
/* critical section */
    sem_wait(m);
    x = x + 1;
    sem_post(m);
    printf("x:%d, i:%d\n",i,x);
    return NULL;
}
```

Nous utilisons le même principe décrit précédemment avec une sémaphore.

Cette fois la sémaphore est initialisée à 1 afin que la ressource soit disponible pour le premier thread.

Le programme va donc créer 10 threads identiques qui vont chacun incrémenter la variable x lorsqu'elle leur sera disponible.

Le résultat est le suivant :

```
corentin@rogstrix15:~/TP3$ ./ex2
x:0, i:1
x:1, i:2
x:2, i:3
x:3, i:4
x:4, i:5
x:5, i:6
x:7, i:7
x:6, i:8
x:8, i:9
x:9, i:10
Final value of x is 10
```

Question4 :

Le but est maintenant d'implémenter un système de producteur consommateur similaire à celui vu en td.

```
main() {
   sprod = CS(v1);
   scons = CS(v2);
   CT(Production);
   CT(Consommation);
   ...
}

Production() {
   while (1) {
      Produire(message);
      P(sprod);
      Déposer(message);
      V(scons);
   }

Consommation() {
   while (1) {
      P(scons);
      Retirer(message);
      V(sprod);
      Consommer(message);
   }
```

La fonction *Produire()* est chargé d'écrire un message dans le tampon de type :*Message k* ou k est incrémenté a chaque message. La fonction *Consommer()* va récupérer le message dans le tampon et l'afficher en majuscules. Les fonctions *Déposer()* et *Retirer()* sont simplement remplacés par la fonction *strcpy(tampon[in], message);* qui permet de stocker un message dans le tampon et à l'inverse de le déposer.

```c
void *Production(void *arg) {
    int i;
    for (i = 0; i < 100; i++) {
        char message[100]; // Message produit par le producteur
        Produire(message); // Production du message
        sem_wait(&sprod); // Attends qu'une des cases du tampon soit libre
        strcpy(tampon[in], message); // Stockage du message dans le tampon
        in = (in + 1) % N;
        sem_post(&scons); // Notification du consommateur qu'un message est disponible dans le tampon
    }
    return NULL;
}
void *Consommation(void *arg) {
    int i;
    for (i = 0; i < 100; i++) {
        char message[100]; // Message consommé par le consommateur
        sem_wait(&scons); // Attend qu'une case du tampon soit pleine
        strcpy(message, tampon[out]); // Récupération du message depuis le tampon
        out = (out + 1) % N;
        sem_post(&sprod); // Notification du producteur qu'une case du tampon est libérée
        Consommer(message); // Consommation du message
    }
    return NULL;
}
```

Etant donné l'utilisation partagée du tampon il est nécessaire d'ajouter des sémaphores qui garantissent qu'on ne produira pas dans une case pleine et/ou en cours de lecture. La taille du tampon étant limité (a 5 messages dans notre cas) le producteur ne doit produire que si une case est libre et le consommateur ne doit lire que si au moins une des cases est pleine. Pour cela il faut obligatoirement que les sémaphores soient initialisées correctement :
-scons, la sémaphore indiquant qu'une lecture est en cours doit être initialisée a 0
-sprod, la sémaphore indiquant qu'un écriture est en cours doit être initialisée à la taille du tampon N (soit 5 dans notre cas).

Le résultat est le suivant :

```
Message produit: Message: 0
Message produit: Message: 1
Message produit: Message: 2
Message produit: Message: 3
Message produit: Message: 4
Message consommé: MESSAGE: 0
Message consommé: MESSAGE: 1
Message consommé: MESSAGE: 2
Message consommé: MESSAGE: 3
Message consommé: MESSAGE: 4
Message produit: Message: 5
Message produit: Message: 6
Message produit: Message: 7
Message produit: Message: 8
Message produit: Message: 9
Message produit: Message: 10
Message produit: Message: 11
Message consommé: MESSAGE: 5
Message consommé: MESSAGE: 6
Message consommé: MESSAGE: 7
Message consommé: MESSAGE: 8
Message consommé: MESSAGE: 9
Message consommé: MESSAGE: 10
Message consommé: MESSAGE: 11
```

TP4 :

Le but de de ce TP est de simuler une course de voiture avec des threads Posix vus dans le TP3.

Le but étant de faire simuler une course ou un arbitre donne le départ avant que les voitures n'effectuent des tours. Chaque voiture met un temps aléatoire pour effectuer le tour et à la fin un classement est donné par l'arbitre.

Version simplifiée :

Dans un premier temps nous ferons une version simplifiée ou l'arbitre se contentera de donner le départ de la course et d'afficher le classement final et ce sont les voitures qui afficheront leurs tours et si elles ont fini.

Nous avons donc défini un fonction *course_voiture(*arg)* qui définit ce que doit faire la voiture. La fonction reçoit un pointeur vers la voiture concerné en argument.

Dans un premier temps la voiture attend l'ordre de départ donné par l'arbitre. Puis elle effectue tous ses tours en affichant un message a chaque tour finit. Lorsque tous les tous ont été effectués la voiture met à jour sa position d'arrivée dans le tableau d'arrivée et affiche un message de fin.

Afin que plusieurs voitures ne finissent pas en même temps il est essentiel de protéger cette section avec un mutex : - *pthread_mutex_lock(&mutex);*

-pthread_mutex_unlock(&mutex);

Cela permet donc qu'il y ait un ordre précis dans le classement des voitures.

Nous avons ensuite une fonction *arbitre_role()* qui définit les actions de l'arbitre. Une fois que toutes les voitures sont prêtes l'arbitre annonce le départ de la course. Il lance ensuite tous les threads créés pour les voitures. Une fois que toutes les voitures ont fini il parcourt la tableau d'arrivée pour afficher le classement final de la course.

Afin que les voitures partent toutes en même temps il en essentiel de créer un barrière qui attend que toutes les voitures soient prêtes. Pour cela nous utilisons dans *course_voiture* et dans *arbitre_role()*:

pthread_barrier_wait(&barriere);

En initialisant le thread comme ceci : *pthread_barrier_init(&barriere, NULL, NUM_CARS + 1);*

La thread attend donc que toutes les voitures et l'arbitre soient prêts avant de lancer la course.

Le résultat est le suivant :

```
corentin@rogstrix15:~/TP4$ ./course
L'arbitre déclenche le départ de la course
Voiture 1 a fini le tour  1
Voiture 2 a fini le tour  1
Voiture 1 a fini le tour  2
Voiture 1 a fini le tour  3
Voiture 1 a fini le tour  4
Voiture 0 a fini le tour  1
Voiture 2 a fini le tour  2
Voiture 1 a fini le tour  5
Voiture 1 a fini la course!
Voiture 2 a fini le tour  3
Voiture 0 a fini le tour  2
Voiture 0 a fini le tour  3
Voiture 2 a fini le tour  4
Voiture 0 a fini le tour  4
Voiture 2 a fini le tour  5
Voiture 2 a fini la course!
Voiture 0 a fini le tour  5
Voiture 0 a fini la course!
Course terminée voici le classement:
1. Voiture 1
2. Voiture 2
3. Voiture 0
```

Version classique

Dans une version moins simplifié nous voulons que l'arbitre mette à jour le nombre de tours de chaque voiture et qu'il affiche le message à chaque tour pour chaque voiture.

Le but étant d'imager un fonctionnement avec plusieurs machines distantes qui pourraient communiquer avec une machine centrale chargée d'afficher les données de chaque machine etc…

Pour cela nous allons toujours gardé le mutex qui permet de définir un ordre d'arrivé. Il est possible que plusieurs voiture finissent leur dernier tour en même temps et envoie un signal a l'arbitre. Il est donc essentiel que la section ou l'arbitre met a jour la position finale de chaque voiture soit protégée.

Cependant il faut maintenant rajouter un mutex a chaque voiture. En effet l'arbitre va observer toutes les voitures et mettre a jour leur nombre de tours etc… Il ne faut donc pas que la voiture complete un autre tour avant que l'arbitre n'ai eu le temps de noter et indiquer les informations sur le tour en cours. Pour cela chaque voiture va prendre en paramètre un mutex et un signal qu'elle appellera a la fin de chaque tour.

La section partagée entre l'arbitre et la voiture est la suivante dans le thread voiture:

```
pthread_mutex_lock(&voiture->mutex);
voiture-> tours_fini++;// Signal pour indiquer la fin du tour
voiture-> tours_completes++;
pthread_mutex_unlock(&voiture->mutex);
```

Et dans le thread arbitre :

```
if (Liste_voitures[i].tours_fini == 1) {//si la voiture a finit son tour en cours

    //nous arrivons dans la section critique
    //nous empechons donc la voiture de continuer son programme tant qe l'arbitre n'a pas noté les temps et les tours de la voitur
    pthread_mutex_lock(&Liste_voitures[i].mutex);
    printf("Voiture%d a fini le tour %d en %f secondes\n", Liste_voitures[i].id, Liste_voitures[i].tours_completes, Liste_voitures
    if (Liste_voitures[i].tours_completes == NUM_LAPS) {//si la voiture a fini la course

        //section critique partagée par toutes les voitures
        pthread_mutex_lock(&mutex);
        position_course++;
        Liste_voitures[i].position = position_course;
        ordre_arrivee[Liste_voitures[i].position - 1] = Liste_voitures[i].id;
        printf("Voiture%d a fini la course!\n", Liste_voitures[i].id);
        nb_voitures_arrivees++;
        pthread_mutex_unlock(&mutex);
        Liste_voitures[i].avg_time=Liste_voitures[i].total_time/NUM_LAPS;
    }

    Liste_voitures[i].tours_fini = 0;//on indique que l'on repart dans un nouveau tour
    pthread_mutex_unlock(&Liste_voitures[i].mutex);//on laisse la voiture refaire un tour
```

L'arbritre observe donc toutes les voitures et si ils voient qu'un voiture a finit un tour il affiche les informations sur le tour correspondant. Si la voiture a terminé il utilise le même procédé que dans la course simplifié pour gérer la fin de course.

Lorsque toutes les voitures sont arrivées l'arbitre sort de sa boucle d'observation pour donner le classement final.

Le système de barrière pour le départ est identique a celui de la course simplifié et tous les mutex sont initialisés à 0.

Le résultat d'une course est le suivant :

```
corentin@rogstrix15:~/TP4$ ./course_normale
L'arbitre déclenche le départ de la course
Voiture0 a fini le tour 1 en 0.750620 secondes
Voiture2 a fini le tour 1 en 1.078299 secondes
Voiture1 a fini le tour 1 en 1.225221 secondes
Voiture0 a fini le tour 2 en 0.560600 secondes
Voiture1 a fini le tour 2 en 0.994962 secondes
Voiture0 a fini le tour 3 en 0.932635 secondes
Voiture2 a fini le tour 2 en 1.343222 secondes
Voiture1 a fini le tour 3 en 0.551527 secondes
Voiture2 a fini le tour 3 en 0.942162 secondes
Voiture0 a fini le tour 4 en 1.275347 secondes
Voiture1 a fini le tour 4 en 0.876530 secondes
Voiture0 a fini le tour 5 en 0.564333 secondes
Voiture0 a fini la course!
Voiture1 a fini le tour 5 en 0.910716 secondes
Voiture1 a fini la course!
Voiture2 a fini le tour 4 en 1.374462 secondes
Voiture2 a fini le tour 5 en 0.509410 secondes
Voiture2 a fini la course!
Course terminée voici le classement:
1. Voiture 0 en 4.083534, secondes avec une moyenne de 0.816707 s/tour
2. Voiture 1 en 4.558956, secondes avec une moyenne de 0.911791 s/tour
3. Voiture 2 en 5.247555, secondes avec une moyenne de 1.049511 s/tour
```

Hormis le temps que j'ai décidé d'implémenter, la sortie est similaire à la course simplifiée. Ce qui change est la façon dont sont annoncés les tours et les temps puisque tout est géré par l'arbitre au lieu de chaque voiture.

Conclusion:

Ce TP nous a permis de découvrir la synchronisation des processus en utilisant des sémaphores et des threads POSIX. Nous avons exploré plusieurs concepts clés, tels que la signalisation, le rendez-vous, l'exclusion mutuelle, la synchronisation producteur-consommateur et enfin la gestion de plusieurs threads afin qu'ils interagissent entre eux.

Ce TP nous a fourni une expérience pratique précieuse pour comprendre la synchronisation des processus et la gestion des threads dans un environnement multithread. En apprenant à utiliser efficacement les sémaphores et les mutex, j'ai pu mieux comprendre comment interagissaient les différents processus d'un système d'exploitation.