TD15

Robot de chimie (agreg)

Augustine Picquet

Dépendances Ce sujet contient trois parties indépendantes qui doivent être traitées toutes les trois. On veillera à bien indiquer sur la copie les changements de partie.

Attendus Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

1 Partie I. Robot de dépôt de solutions chimiques

Cette partie doit être traitée dans le langage C avec des appels de type POSIX. Un rappel de certaines fonctions est disponible en annexe. Aucun `#include`n'est demandé. Ce sujet s'intéresse principalement à la synchronisation de processus légers (appelés dans la suite threads) à l'aide de mutex et de sémaphores. Les trois sections sont indépendantes.

On s'intéresse à la partie contrôle d'un ou plusieurs robots d'injections de produits chimiques dans des éprouvettes. On considère avoir un bras injecteur contrôlable par un ou plusieurs threads sur un ordinateur connecté à ce bras. Dans la suite, on supposera avoir une bibliothèque de fonctions dont on donnera la spécification mais dont on supposera ne pas pouvoir modifier ni consulter le code.

On s'intéresse à résoudre les problèmes découlant du cas où plusieurs threads d'un programme interagissent avec le robot au travers de la bibliothèque de contrôle en même temps. On considère exister la structure struct formule.

Dans l'ensemble du sujet, on suppose avoir un seul processeur avec un seul cœur.

1.1 Une rangée d'éprouvettes

On suppose dans cette partie être dans le cas où le programme composé de plusieurs threads contrôle un seul bras pouvant se déplacer au dessus d'une rangée unique de N>0 éprouvettes.

On suppose avoir trois fonctions appelée, `int libre()`, `void aller(int i)` et `void injecter(struct formule *f)` qui respectivement permettent d'obtenir l'indice d'une éprouvette vide, au bras d'aller à l'éprouvette i entre 0 inclus et N exclus, et enfin d'injecter une solution dont la formule est passée en argument dans l'éprouvette actuellement sous le bras que l'on considère vide. Dans cette section on considère avoir assez d'éprouvettes par rapport au nombre d'expériences prévues.

Le comportement d'une fonction utilisant le bras sera alors par exemple (sous l'hypothèse qu'il y a toujours au moins une éprouvette vide disponible) :

```
1 int deposer(struct formule *f) {
2     int i = libre();
3     aller(i);
4     injecter(f);
5     return i;
6 }
```

Question 1.1 On se place dans le cas où deux threads tentent « en même temps » d'utiliser la fonction déposer décrite ci-dessus. Explicitez un exemple montrant qu'il est possible que les deux threads déposent leur solution dans la même éprouvette.
Corrigé : Si `libre` permet d'avoir deux indices différents i et j, alors les deux threads vont se déplacer et injecter tous les deux dans le même éprouvette. En effet, c'est le cas si l'exécution s'exécute ainsi :

thread 1	thread 2
i = libre()	.
.	i = libre()
aller(i)	.
injecter(i)	.
return i ;	.
.	aller(i)
.	injecter(i)
.	return i

Question 1.2 On tente de résoudre le problème avec des mutex

```
1  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3  int deposer(struct formule *f) {
4      pthread_mutex_lock(&mutex);
5      int i = libre();
6      pthread_mutex_unlock(&mutex);
7
8      aller(i);
9      injecter(f);
10     return i;
11 }
```

Quel est le problème de cette solution ?
Corrigé : Il est toujours possible de faire la même chose que ci-dessus en commençant par verrouiller puis en appelant libre puis en déverrouillant, en passant la main. Dès lors le second thread va lui aussi verrouiller, appeler libre et avoir le même indice que le premier thread, les deux vont donc injecter dans la même éprouvette.

Question 1.3 On tente de corriger le problème de la solution précédente comme suit :

```
1  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3  int deposer(struct formule *f) {
4      pthread_mutex_lock(&mutex);
5      int i = libre();
6      aller(i);
7      injecter(f);
8      return i;
9      pthread_mutex_unlock(&mutex);
10 }
```

Cette version fonctionne-t-elle ? Expliquez pourquoi et corrigez-la si elle ne fonctionne pas.
Corrigé : Non, cette version ne fonctionne pas. Le premier thread va verrouiller et ne pourra jamais déverrouiller, car il va sortir de la fonction `deposer` avant de déverrouiller le verrou.
Pour résoudre le problème, il suffit d'intervertir les lignes 8 et 9.

Question 1.4 On suppose, pour cette question uniquement, que la formule est à encoder pour le modèle spécifique de bras : la fonction `injecter` n'est pas capable de directement traiter la formule de type `struct formule`. La fonction `encoder` permet de traduire la formule pour le bras utilisé. Le code de base deviendrait :

```
1  int deposer(struct formule *f) {
2      int i = libre();
3      aller(i);
4      struct codage code = encoder(f, bras_modele_1);
5      injecter(&code);
6
7      return i;
8  }
```

On considère aussi que la fonction (sans effets de bords) encoder est très lente, beaucoup plus lente que le temps de déplacement du bras. Modifiez le code précédent pour qu'il soit possible par plusieurs threads de l'appeler en parallèle sans problème de synchronisation et que ces appels soient le plus efficaces possibles.
Corrigé : Dans cette version, la fonction qui prend beaucoup de temps est en dehors du verrou, ce qui permet une bien meilleure parallélisation des tâches. Par ailleurs, les opérations critiques sont bien exécutées avec un verrou verrouillé, ce qui garantit l'absence de problème de concurrence.

```
1  int deposer(struct formule *f) {
2      struct codage code = encoder(f, bras_modele_1);
3      pthread_mutex_lock(&mutex);
4      int i = libre();
5      aller(i);
6      injecter(&code);
7      pthread_mutex_unlock(&mutex);
8      return i;
9  }
```

1.2 Ajout du vidage des éprouvettes

On considère maintenant avoir la fonction `struct resultat *analyser()` qui permet de vider et d'analyser l'éprouvette qui est sous le bras. Le code d'un thread complet serait alors (en considérant que le mutex `mutex` a été créé en tant que variable globale) :

```
...
struct formule f = creer_formule (..);
int pos = deposer(&f);
... // attente avec calcul du temps de deplacement du bras
pthread_mutex_lock(&mutex);
aller(pos)
struct resultat *res = analyser ()
pthread_mutex_unlock(&mutex);
...
```

Question 1.7 On suppose avoir maintenant plus d'expériences à faire que d'éprouvettes. On propose la modification suivante de la fonction `deposer`. On considère toujours avoir un mutex `mutex` global initialisé. On considère aussi qu'après que la fonction `analyser` termine de vider l'éprouvette, la fonction `libre` est capable de renvoyer sa position. Lorsqu'il n'y a pas d'éprouvette libre, la fonction libre renvoie la valeur (-1). On transforme la fonction `deposer` :

```
int deposer(struct formule *f) {
    int i = -1;
    while (i == -1){
        pthread_mutex_lock(&mutex);
        i = libre();

        if (i != -1){
            aller(i);
            injecter(f);
        }
        pthread_mutex_unlock(&mutex);
    }
    return i;
}
```

Est-ce que cela fonctionne? Justifier votre réponse

Corrigé : Oui, cela fonctionne.

- Exclusion mutuelle :
  Si un thread veut entrer, alors il verrouille. S'il y arrive, c'est qu'il n'y a aucun thread dans la section critique, il peut donc faire ce qu'il a faire, puis déverrouiller quand il sort de la section critique OK.
  Donc deux thread ne peuvent entrer en section critique en même temps, on a bien l'exclusion mutuelle.
- Interblocage :
  Le thread va bien déverrouiller quand il va sortir de section critique, donc un autre thread pour entrer en section critique à ce moment là. On a bien l'absence d'interblocage.
- Absence de famine :
  Validée par l'absence d'interblocage.

Question 1.8 En utilisant la fonction `deposer` de la question précédente, explicitez quel impact négatif a cette approche sur les autres threads et/ou les autres processus

Corrigé : Quand un thread demande à entrer en section critique, alors il bloque tous les autres threads qui ne peuvent faire autre chose en parallèle, ils ne peuvent pas déposer quoi que ce soit, ce qui est pas optimal pour la parallélisation.

Question 1.9 On suppose maintenant avoir accès aux sémaphores, et il n'y a plus accès qu'aux fonction `aller`/`injecter`/`analyser` (ie plus à la fonction `libre`). Proposer une implémentation en C des fonctions *deposer* et *recuperer* en précisant les arguments et en justifient ceux éventuellement ajoutés. Le but est d'autoriser l'utilisation suivante :

```
???
struct  formule f = creer_formule(...)
int pos = deposer(&f, ???)
///attendre en fonction de la formule
struct resultat *res recupere(pos, ???)
???
```

Précisez les variables partagées utilisées et l'initialisation du sémaphore dans la fonction main si besoin. On s'attachera à ce que l'implantation proposée résolve le problème soulevé dans la question précédente. On considérera avoir accès à une fonction `int trouver(int *tab, int n)` qui trouve et renvoie la valeur de l'indice dans le tableau tab de n entiers qui a pour valeur 0. Si aucune valeur n'est 0 alors la fonction renvoie -1.
Corrigé : On va ajouter un sémaphore tel que quand une éprouvette est libérée alors on augmente sa valeur d'un. On considère que toutes les éprouvettes sont initialement vides, donc le sémaphore sera initialisé à N.
On utiliser un tableau de taille N (variable globale) qui va contenir 0 en case i si l'éprouvette i est libre, 1 sinon.
On utilise un verrou m pour protéger les lectures/écritures dans le tableau et le lien entre aller et les opérations injecter/analyser.

```
int deposer(struct formule *f){
    sem_wait(&s);
    pthread_mutex_lock(&m);
    int i = trouver(tableau, N);
    tab[i] = 1;
    aller(i);
    deposer(&f);
    pthread_mutex_unlock(&m);
    return i;
}

struct resultat* recuperer(int pos){
    pthread_mutex_lock(&m);
    tab[pos] = 0;
    aller(pos);
    struct resultat *r = analyser();
    pthread_mutex_unlock(&m);
    sem_post(&s);
    return r;
}
```

1.3 Passage à la 2D

Pour augmenter le nombre d'expériences possibles en même temps, on ajoute plusieurs rangées d'éprouvettes (toutes de la même taille), tout en gardant un seul bras. On ajoute un moteur permettant de changer de rangée d'éprouvettes. On a donc maintenant deux fonctions indépendantes `allerX` et `allerY`. On a respectivement maintenant NY rangées de NX éprouvettes. La fonction `libreXY` modifie ses arguments pour renvoyer les coordonnées d'un emplacement vide au moment de son appel. Dans cette partie, on suppose à nouveau avoir assez d'emplacement pour l'ensemble des expériences.

```
void deposer(struct formule *f){
    int x,y;
    libreXY(&x,&y);
    allerX(x);
    allerY(y);
    injecter(f);
}
```

Question 1.11 Combien de mutex au minimum sont nécessaires pour permettre l'utilisation de la fonction `deposer` par plusieurs threads du même programme en garantissant uniquement le bon fonctionnement du système (ie on ne s'intéressera pas à la problématique de performance dans cette question). Modifier le code précédent pour ajouter ce ou ces mutex.
Corrigé : 1 seul mutex (nommé ici `deplacement`) suffit, qui verrouille tous les déplacements (ie les déplacements selon toutes les coordonnées) .En modifiant le code, on aboutit à :

```
void deposer(struct formule *f){
    int x,y;
    pthread_mutex_lock(&deplacement);
    libreXY(&x,&y);
    allerX(x);
    allerY(y);
    pthread_mutex_unlock(&deplacement);
    injecter(f);
}
```

Question 1.12 Les deux moteurs sont indépendants, plutôt que d'attendre le déplacement sur X soit fini avant de faire le déplacement sur Y, il est possible de faire les deux en même temps. Proposez une implémentation de la fonction déposer réalisant cette amélioration.
Corrigé : Notons que nous devons créer la fonction`aller_bisX` car pour appeler le mutex avec une fonction, il faut qu'elle soit de type void*-> void*. On la fonction donnée par l'énoncé est de type int->void*, qui ne convient pas.

```
void* aller_bisX(void* arg){
    int* x = (int*) arg;
    allerX(*x);
    return NULL;
}

void deposer(struct formule* f){
    int x,y;
    pthread_mutex_lock(&m);
    libreXY(&x,&y);
    pthread p;
    pthread_create(&p, NULL, aller_bisX, &x);
    allerY(y);
    pthread_join(p);
    injecter();
    pthread_mutex_unlock(&m);
}
```

1.4 Analyse non destructive dans le cas d'une seule rangée

On change légèrement l'expérience tout en gardant le même matériel. Les expérimentateurs mettent des produits dans chacune des éprouvettes et modifient légèrement le matériel pour que l'analyse ne vide plus les éprouvettes. Les expérimentateurs veulent étudier l'évolution du produit dans le temps et vont donc appeler la fonction `analyser` régulièrement. On revient dans le cas où l'on a une seule range et on considère que les éprouvettes sont déjà remplies au lancement du programme.

```
...
while(true){
    aller(i)
    struct resultat *res = analyser ()
    traiter(res, thread_data);
    free(res);
}
```

La fonction `traiter` utilise uniquement des données internes à un thread (la variable `thread_data`) et donc il n'y a aucun problème à en exécuter plusieurs en même temps.

On se rend compte que le moteur est assez lent, mais est 10 fois plus rapide pour aller en valeurs croissantes que décroissantes. Par exemple, en étant en position 1, `aller(2)` est dix fois plus rapide que `aller(0)`.

Question 1.13 Expliciter le déplacement de la tête dans le cas où les évènement suivants se produisent : '

- L'analyse de 5 est en cours et à commencé à T0 (qui finira a T0+6)
- Un autre thread lance à T0+1 l'analyse de 3 (qui prendra 1)
- Un autre thread lance à T0+2 l'analyse de 7 (qui prendra 4)
- Un autre thread lance à T0+8 l'analyse de 9 (qui prendra 1)

Toute les durées sont en Unité de Temps (UT). On donnera l'ordre des actions ainsi que l'UT correspondant par rapport à T0 (donc T0+3 sera donné comme 3). On considère que déplacer le bras d'une position en valeur croissante prend une UT. On considère aussi que l'on a déjà réglé le problème de synchronisation entre les threads.
Corrigé :

T0+6	fin analyse de 5
T0+26	début analyse de 3
T0+27	fin analyse 3
T0+31	début analyse 7
T0+35	fin de l'analyse de 7
T0+37	début de l'analyse de 9
T0+38	fin de l'analyse de 9

Question 1.14 Ecrire une fonction `analyser_en(i)` qui garantit que si plusieurs analyses sot en attente, celle qui sera débloquée sera soit cette dont la position est la proche supérieurement si elle existe, soit cette dont la position est la plus petite sinon.

Corrigé : On utilise un tableau want qui va indiquer en position i si on a besoin d'aller analyser l'éprouvette i (par un booléen, un entier, ou encore par un sémaphore initialisé à 1)
On utilisera aussi un mutex m.
NB : cette correction a été faite en classe donc nous la présupposons comme valide. Néanmoins à mes yeux, elle n'a pas l'air de répondre à la question

```
struct resultat* analyser_en(int i){
    int seul = 1;
    pthread_mutex_lock(&m);
    for(int j= 0; j<n ;j++){
        if(i!=j && want[j]){ //si on a une autre eprouvette que la i-eme dont il faut s'occuper
            seul = 0;
        }
    }
    want[i] = 1;
    pthread_mutex_unlock(&m);
    if(seul == 1){//si la seule eprouvette dont il faut s'occuper est la i-eme
        aller(i);
        struct resultat* r = analyser();
        pthread_mutex_lock(&m);
        want[i] = 0;
        pthread_mutex_unlock(&m);
    }
    else{
        sem_wait(&t[i]);
        aller(i);
        struct resultat* r = analyser();
        pthread_mutex_lock(&m);
        want[i] = 0;
        pthread_mutex_unlock(&m);
    }
    if(want[i]){
        sem_post(&t[i]);
    }
}
```

Cette réponse ci me semble répondre bien mieux à la question posée. Rien dans le sujet ne semble présupposer que l'éprouvette i soit à analyser...

```
struct resultat* analyser_en(int i){
    int ind = -1;
    pthread_mutex_lock(&m);
    for(int j= i; j<n ;j++){
        if(i!=j && want[j]&& ind == -1){ //si on a une autre eprouvette plus grande que la i-eme dont il
            faut s'occuper
            ind = j;
        }
    }
    want[i] = 1;
    pthread_mutex_unlock(&m);
    if(ind != -1){//s'il y a une eprouvette plus grande dont il faut s'occuper
        aller(ind);
        struct resultat* r = analyser();
        pthread_mutex_lock(&m);
        want[ind] = 0;
```

```
        pthread_mutex_unlock(&m);
        return r;
    }
    else{
        for(int j= 0; j<=i; j++){
            if(i!=j && want[j]&&ind == -1){ //si on a une autre eprouvette plus petite que la i-eme dont il
                faut s'occuper
            ind = j;
            }
        }
        if(ind != -1){
            aller(ind);
            struct resultat* r = analyser();
            pthread_mutex_lock(&m);
            want[ind] = 0;
            pthread_mutex_unlock(&m);
            return r;
        }
        else{
            return NULL;
        }
    }

}
```

2 Annexe

```
pthread_create // Creer un nouveau thread
int pthread_create(pthread_t * thread,
        pthread_attr_t * attr,
        void * (*start_routine)(void *),
        void * arg);

pthread_join // Attendre la fin d un autre thread
int pthread_join(pthread_t th, void **thread_return);

pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock
        , pthread_mutex_destroy // Operations sur les mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

sem_init // Initialiser un semaphore non nomme
int sem_init(sem_t *sem, int pshared, unsigned int value);

sem_destroy // Detruire un semaphore non nomme
int sem_destroy(sem_t *sem);

sem_post // Deverrouiller un semaphore
int sem_post(sem_t *sem);

sem_wait // Verrouiller un semaphore
int sem_wait(sem_t *sem);
```