TP 3 : THREAD POSIX
Philosophes, producteurs-consommateurs, etc.

SEPC : Ensimag 2A

Résumé

L'objectif du TP est la programmation d'un problème de synchronisation entre des processus légers (threads) à l'aide de la bibliothèque de threads POSIX. Nous avons étudié en travaux dirigés quelques problèmes standards. Vous devez implanter une solution combinant deux de ces problèmes et respectant quelques contraintes. Une des synchronisations sera réalisée avec des moniteurs POSIX, tandis que l'autre sera réalisée avec des sémaphores.

C'est la qualité des implantations des synchronisations qui sera déterminante dans la notation.

1 Le sujet

Votre programme devra donc implanter la combinaison de deux problèmes standards. L'un des problèmes sera implanté avec des moniteurs, l'autre avec des sémaphores.

Attention, le sujet est imposé ! L'implantation d'une solution à un autre problème que celui demandé, ou l'utilisation de moniteurs à la place de sémaphores, et vice versa, sera considérée comme un hors sujet, noté en conséquence.

1.1 Les variantes

Le sujet est fonction de votre numéro d'utilisateur (UID, obtenu avec la commande `id -u`) modulo le nombre de sujet. C'est l'UID minimum qui est utilisé pour un binôme ou un trinôme. Pour connaître le numéro de votre sujet, il suffit d'exécuter les commandes suivantes sur votre serveur de référence (ensibm).

– Pour un monôme lançant lui-même la commande

```
echo "$(id -u) % 15" | bc
```

– Pour un binôme ayant pour identifiants login1 et login2

```
echo "define min(x,y) { if (x<y) return x else return y }; \
          min( $(id -u login1), $(id -u login2)) % 15" | bc
```

Par exemple, le sujet numéro 7 consiste à implanter le problème des lecteurs-rédacteurs avec des moniteurs et un allocateur avec des sémaphores.

	Implantation en moniteur			
Impl. en Sémaphores	Philo. (états)	Lecteur-Rédacteur	Allocateur	Producteur-conso.
Philo. (états)		0	1	2
Lecteur-Rédacteur	3		4	5
Allocateur	6	7		8
Producteur-conso.	9	10	11	
Philo. (gaucher)		12	13	14

TABLE 1 – Les variantes des sujets

2 Les sujets

Chaque sujet implante un certain nombres de fonctions de synchronisation. ces fonctions sont appelées par des threads décrits dans la section 3.

2.1 Philosophes

Chaque philosophe est identifié par un numéro (i). Les fonctions de synchronisations sont

```
void prendre_fourchettes(int i);
void poser_fourchettes(int i);
```

Il y a une seule classe de threads.

2.2 Lecteur-rédacteurs

```
void debut_lire();
void fin_lire();
void debut_ecrire();
void fin_ecrire();
```

Il y a deux classes de threads séparés.

2.3 Producteur-consommateur

La taille du tampon géré sera égale à 2.

```
void deposer(int message);
int retirer();
```

Il y a deux classes de threads séparés.

2.4 Allocateur

Le nombre de ressources totales sera égale à 10.

```
void allouer(int n);
void liberer(int n);
```

Il y a une seule classe de threads.

3 Combinaison des problèmes

8 threads seront lancés qui exécuteront chacun 10 fois une boucle de calcul. Du coup, suivant les combinaisons, ces threads seront groupés en 1, 2 ou 4 classes.

La boucle de base d'un thread sera la suivante :

```
void *thread(void *)
{
  for(i=0; i < 10; i++)
  {
    // fonctions liees au probleme en moniteur

    usleep( 100 * drand48() ); // un peu d'attente aleatoire

    // fonctions liees au probleme en semaphore
  }
}
```

3.1 Exemples

3.1.1 Allocateur en moniteur et des philosophes en sémaphores (sujets 1 et 13)

Il n'y a qu'une seule boucle.

```
void *thread1(void *a)
{
  int id = (int) a;
  for(int i=0; i < 10; i++)
  {
    int v = 10 * drand48();
    allouer(v);
    liberer(v);
    usleep(100 * drand48()))
    prendre_fourchette(i);
    poser_fourchette(i);
  }
}
```

3.1.2 Philosophes en moniteurs et un producteur-consommateur en sémaphore (sujet 9)

Les 2 codes des boucles seront donc

```
void *thread1(void *a)
{
  int id = (int) a;
  for(int i=0; i < 10; i++)
  {
    prendre_fourchette(id)
```

```
void *thread2(void *a)
{
  int id = (int) a;
  for(int i=0; i < 10; i++)
  {
    prendre_fourchette(id)
```

```
    poser_fourchette(id);
    usleep(100 * drand48());
    deposer(i);
  }
}
```

```
    poser_fourchette(id);
    usleep(100 * drand48());
    int v = retirer(i);
  }
}
```

3.1.3 Producteur-consommateur en moniteur et un lecteur-rédacteur en sémaphore (sujet 5)

Il y a 4 boucles.

```
void *thread1(void *a)
{
  int id = (int) a;
  for(int i=0; i < 10; i++)
  {
    deposer(id);
    usleep(100 * drand48());
    debut_lire();
    fin_lire();
  }
}
void *thread3(void *a)
{
  int id = (int) a;
  for(int i=0; i < 10; i++)
  {
    deposer(id);
    usleep(100 * drand48());
    debut_ecrire();
    fin_ecrire();
  }
}
```

```
void *thread2(void *a)
{
  int id = (int) a;
  for(int i=0; i < 10; i++)
  {
    retirer(id) ;
    usleep(100 * drand48());
    debut_lire();
    fin_lire();
  }
}
void *thread4(void *a)
{
  int id = (int) a;
  for(int i=0; i < 10; i++)
  {
    retirer(id) ;
    usleep(100 * drand48());
    debut_ecrire();
    fin_ecrire();
  }
}
```

Il faut, bien sûr, ajouter des structures de données et les initialiser correctement. Cela inclut les variables mutex, variables de conditions, sémaphores, les tampons, les indices de lecture ou d'écriture dans le tampon, ainsi que tout ce qui vous semblera nécessaire. Il faudra lancer les threads en parallèles et leur faire exécuter les fonctions. Il faudra aussi gérer correctement la terminaison.

Des traces pertinentes (printf) devront montrer les synchronisations. Vous pouvez modifier les constantes (constantes entières, nombre de tours, nombre de threads, taille du tampon) pour illustrer des points particuliers pertinents.