Synchronisation de Threads avec les Sémaphores POSIX

Afin de comprendre l'intérêt des sémaphores, observez les résultats pour le moins bizarres fournis par le code en annexe (c*ompt.c)*. Ce programme crée deux threads qui incrémentent chacun, **N** fois, une variable globale **compteur**.

Exercice 1. Créez un répertoire nommé semposix dans votre répertoire ASR4 puis copiez le fichier c*ompt.c*. Compilez ce programme comme ceci :

```
gcc compt.c -o compt -lpthread
```

Exécutez plusieurs fois *compt.* Comme compteur est initialisé à zéro et que chacun des deux threads l'incrémente N fois, **compteur** devrait valoir **2*N** à la fin du programme. Observez la valeur de **compteur**, comment expliquez vous ce comportement ?

Les threads peuvent grandement améliorer l'écriture de programmes efficaces et élégants. Cependant, des problèmes peuvent apparaître lorsque plusieurs threads partagent des adresses mémoires communes.

Pour comprendre ce qui se passe, analysons la portion de code suivant :

```
Thread 1            Thread 2
 a = data;          b = data;
 a++;               b--;
 data = a;          data = b;
```

Si ce code est exécuté séquentiellement d'abord par Thread1 puis par Thread2, il n'y a aucun problème est le résultat obtenu est conforme à notre attente. Mais les threads s'exécutent en parallèle et dans un ordre quelconque. Imaginez la situation suivante :

Thread 1	**Thread 2**	**data**
a = data;	---	0
a = a+1;	---	0
---	b = data; // 0	0
---	b = b - 1;	-1
data = a; // 1	---	1
---	data = b; // 1	-1

La variable **data** peut finir avec la valeur +1, 0, -1, et il n'y a aucun moyen de connaître avec certitude la valeur finale.

Une solution serait d'empêcher tout autre thread d'accéder à la variable data à partir du moment où un thread y accède déjà. Les sémaphores permettent de réaliser ce blocage.

Les semaphores Posix

Toutes les fonctions et types des sémaphore POSIX sont prototypés ou définis dans semaphore.h.
Pour définir un sémaphore utilisez :

```
sem_t sem_name;
```

Pour initialiser un semaphore utilisez sem_init:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- sem pointe vers un objet semaphore à initialiser
- pshared est un flag qui indique si oui ou non le sémaphore peut être partagé par les processus fork()és. Les Threads linux ne supportent pas pour l'instant les sémaphores partagés.
- value est la valeur initiale du sémaphore

Exemple :

```
sem_init(&sem, 0, 1);
```

Pour attendre la libération d'un sémaphore, utilisez sem_wait:

```
int sem_wait(sem_t *sem);
```

Exemple:

```
sem_wait(&sem);
```

- Si la valeur du sémaphore est négative, le processus appelant est bloqué. Un processus bloqué sera réveillé lorsqu'un processus appelle sem_post.

Pour incrémenter la valeur d'un semaphore, utilisez sem_post:

```
int sem_post(sem_t *sem);
```

Exemple:

```
sem_post(&sem);
```

- Incrémente la valeur du sémaphore et réveille un processus bloqué en attente du sémaphore s'il y en a un.

Pour connaître la valeur d'un sémaphore utilisez :

```
int sem_getvalue(sem_t *sem, int *valp);
```

- récupère la valeur courante de sem et la place dans la variable pointée par valp

Exemple:

```
int valeur;
sem_getvalue(&sem, &valeur);
printf("Valeur du semaphore %d\n", valeur);
```

Pour détruire un sémaphore :

```
int sem_destroy(sem_t *sem);
```

- détruit le sémaphore ; la destruction n'est possible que si aucun thread n'est en attente de ce sémaphore.

Exemple:

```
sem_destroy(&sem);
```

Utilisation des sémaphores

Reprenons l'exemple précédent

Déclarez le sémaphore en variable globale (or de toute fonction):

```
sem_t mutex;
```

Initialisez le sémaphore dans la fonction main :

```
sem_init(&mutex, 0, 1);
```

Thread 1	**Thread 2**	**data**
sem_wait (&mutex);	---	0
---	sem_wait (&mutex);	0
a = data;	/* blocked */	0
a = a+1;	/* blocked */	0
data = a;	/* blocked */	1
sem_post (&mutex);	/* blocked */	1
/* blocked */	b = data;	1
/* blocked */	b = b - 1;	1
/* blocked */	data = b;	2
/* blocked */	sem_post (&mutex);	2
[data est correct et accessible]		

Exercice 2. Utilisez l'exemple ci-dessus pour corriger le programme de l'annexe, il doit fournir une sortie égale à 2*N.

Pour compiler un programme utilisant les threads et les sémaphores posix:

```
gcc -o filename filename.c -lpthread -lrt
```

Annexe

compt.c :

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define N 1000000

int compteur = 0;

void * incr(void * a)
{
   int i, tmp;
   for(i = 0; i < N; i++)
   {
      tmp = compteur;        /* copie la variable globale compteur localement */
      tmp = tmp+1;           /* incrémente la copie locale */
      compteur = tmp;        /* stocke la valeur locale dans la variable globale compteur */
   }
}

int main(int argc, char * argv[])
{
   pthread_t tid1, tid2;

   if(pthread_create(&tid1, NULL, incr, NULL))
   {
     printf("\n ERREUR création thread 1");
     exit(1);
   }

   if(pthread_create(&tid2, NULL, incr, NULL))
   {
     printf("\n ERROR création thread 2");
     exit(1);
   }

   if(pthread_join(tid1, NULL))        /* Attente de fin de thread 1  */
   {
     printf("\n ERREUR  thread 1 ");
     exit(1);
   }

   if(pthread_join(tid2, NULL))        /* Attente de fin de thread 2 */
   {
     printf("\n ERREUR  thread 2");
     exit(1);
   }

   if ( compteur < 2 * N)
      printf("\n BOOM! compteur = [%d], devrait être  %d\n", compteur, 2*N);
   else
      printf("\n OK! compteur = [%d]\n", compteur);

   pthread_exit(NULL);
   return 0 ;
}
```