Faculté des Sciences & Techniques — Université de Limoges

Licence 3ème année

Programmation Concurrente

Corrections – TD n°2

Processus de poids léger ou « threads » & Sémaphore

Processus de poids léger

1 – Soit le programme suivant, commentez son exécution :

```
1  void print_message_function( void *ptr );
2  int main()
3  {
4        pthread_t thread1, thread2;
5        char *message1 = "Hello";
6        char *message2 = "World";
7
8        pthread_create(&thread1, NULL, (void *) &print_message_function, (void *) message1);
9        pthread_create(&thread2, NULL, (void *) &print_message_function, (void *) message2);
10       exit(0);
11 }
12 void print_message_function( void *ptr )
13 {
14       char *message;
15        message = (char *) ptr;
16        printf("%s ", message);
17 }
```

Est-ce que l'affichage va se produire au final ?
Non, car la fonction « main » devient automatiquement une thread qui s'exécute en parallèle également et dont le travail consiste à arrêter **tout de suite** *le processus.*

Remarque : lorsque le processus s'arrête (arrêt de la thread associée à la fonction `main()` *ou appel à* `exit()`*, toutes les threads qui le constituent s'arrêtent.*

2 – Voici une version améliorée du programme précédent, quels sont ces améliorations ?

```
1  void print_message_function( void *ptr );
2  int main()
3  {
4     pthread_t thread1, thread2;
5     char *message1 = "Hello";
6     char *message2 = "World";
7     pthread_create(&thread1, NULL, (void *) &print_message_function, (void *) message1);
8     sleep(10);
9     pthread_create(&thread2, NULL, (void *) &print_message_function, (void *) message2);
10    sleep(10);
11    exit(0);
12 }
13 void print_message_function( void *ptr )
14 {
15    char *message;
16    message = (char *) ptr;
17    printf("%s", message);
18    pthread_exit(0);
19 }
```

L'affichage se réalise bien dans le bon ordre. Mais il est nécessaire d'attendre 20 secondes pour obtenir ce résultat.
Cette solution est à rejeter car il n'est pas facile d'évaluer le temps nécessaire à une thread pour réaliser son travail : la meilleure solution ? Que ce soit la thread elle-même qui informe de la fin de son travail.

▬▬▬ Sémaphore

3 – Il est nécessaire de réaliser une « barrière de synchronisation » :
- ▷ on crée une sémaphore ;
- ▷ on libère cette sémaphore à la fin de chaque thread devant se réalisée avant la dernière ;
- ▷ dans la dernière thread, on prend la sémaphore suivant un nombre de fois égal au nombre de threads.

```
 1 #include <stdio.h>
 2 #include <semaphore.h>
 3 #include <pthread.h>
 4
 5 sem_t sem_thread_barriere;
 6
 7 void *travail_thread(void *args)
 8 {
 9    int rang = *((int *) args);
10
11    printf("Fin du travail de la thread %d\n", rang);
12    sem_post(&sem_thread_barriere);
13 }
14
15
16 int main()
17 {
18    pthread_t id_threads[5];
19    int i, rangs[5];
20
21    sem_init(&sem_thread_barriere, 0, 0);
22    for (i=0; i < 5; i++)
23    {
24       rangs[i] = i;
25       pthread_create(&id_threads[i], NULL, travail_thread, (void *)(&rangs[i]));
26    }
27
28    for (i=0; i < 5; i++)
29       sem_wait(&sem_thread_barriere);
30    printf("Travail final\n");
31 }
```

4 – On alterne entre les deux threads à l'aide d'une sémaphore associée à chaque thread :

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

sem_t sem_thread1;
sem_t sem_thread2;

void *travail_thread1(void *args)
{
   int i;

   for(i=0; i < 3; i++)
   {
      sem_wait(&sem_thread1);
      printf("Travail %d de la thread 1\n", i);
      sem_post(&sem_thread2);
   }
}

void *travail_thread2(void *args)
{
   int i;

   for(i=0; i < 3; i++)
   {
      sem_wait(&sem_thread2);
      printf("Travail %d de la thread 2\n", i);
      sem_post(&sem_thread1);
   }
}

int main()
{
   pthread_t id_threads[2];
   int i;

   sem_init(&sem_thread1, 0, 1);
   sem_init(&sem_thread2, 0, 0);
   pthread_create(&id_threads[0], NULL, travail_thread1, NULL);
   pthread_create(&id_threads[1], NULL, travail_thread2, NULL);
   for(i=0; i < 2; i++)
   {
      pthread_join(id_threads[i], NULL);
   }
   printf("Travail termine\n");
}
```

5 – Écrire le programme basé sur l'utilisation des threads, permettant pour deux threads distinctes de communiquer l'une vers l'autre.

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

typedef struct
{   sem_t semaphore;
   int entree;
   int sortie;
} variables_thread;

sem_t sem_thread_com;
variables_thread tv1, tv2;

void *travail_t1(void *args)
{   int compteur = 0;

   while(compteur <10)
   {
      /* Envoi de la valeur du compteur a T2 */
      tv1.sortie = compteur;
      sem_post(&sem_thread_com);
      sem_wait(&tv1.semaphore);
      /* Reception de la valeur du compteur depuis T2 */
      sem_post(&sem_thread_com);
      sem_wait(&tv1.semaphore);
      compteur = tv1.entree;
      /* Affiche les valeurs de compteur */
      printf ("Compteur = %d\n", compteur);
   }
   exit(0);
}

void *travail_t2(void *args)
{   int val;

   while(1)
   {
      /* Reception de la valeur depuis T1 */
      sem_post(&sem_thread_com);
      sem_wait(&tv2.semaphore);
      val = tv2.entree;
      /* incrementation de val */
      val++;
      /* Envoi de la valeur vers T1 */
      tv2.sortie = val;
      sem_post(&sem_thread_com);
      sem_wait(&tv2.semaphore);
   }
}

void *travail_thread_com(void *args)
{
   while(1)
   {   sem_wait(&sem_thread_com);
      sem_wait(&sem_thread_com);
      tv1.entree = tv2.sortie;
      tv2.entree = tv1.sortie;
      sem_post(&tv1.semaphore);
      sem_post(&tv2.semaphore);
   }
}
int main()
{
   pthread_t id_t1, id_t2;
   sem_init(&tv1.semaphore, 0, 0);
   sem_init(&tv2.semaphore, 0, 0);
   sem_init(&sem_thread_com, 0, 0);
   pthread_create(&id_t1, NULL, travail_t1, NULL);
   pthread_create(&id_t2, NULL, travail_t2, NULL);
   travail_thread_com(NULL);
}
```