Correction TP N°4

Les Threads

Les Threads :

Le gros avantage lié à la notion de processus léger (thread) est un allégement des opérations de commutations de contexte (le contexte mémoire est le même). De même, l'opération de création d'un nouveau fil d'exécution est sensiblement allégée puisqu'elle ne nécessite plus la duplication complète de l'espace d'adressage du processus père.
Or comme les processus légers au sein d'un même processus partagent le même espace d'adressage, il s'ensuit des problèmes de partage des ressources importants qu'on peut gérer avec les mutex et les sémaphores.

Exercice 1 :

Soit le programme suivant :

```
#include <stdio.h>
#include <pthread.h>
void *task1 (void *arg )
{ printf (" Thread 1 \n" );
pthread_exit (0);
return NULL;
}
void *task2 (void *arg )
{ printf (" Thread 2 \n");
pthread_exit (0);
return NULL;
}
int main ( )
{
pthread_t t1, t2;
printf ("main Init\n");
pthread_create (&t1, NULL, task1, NULL);
pthread_create (&t2, NULL, task2, NULL);
printf ("main Fin\n");
return 0;
}
```

1. Tester ce programme, l'ordre d'affichage est-il déterministe ?
   NB : Pour exécuter, il faut ajouter -lpthread ou -pthread (gcc ex1.c -o ex1 -lpthread).
   Lors de l'exécution, l'ordre d'affichage n'est pas déterministe et cela dépend de l'ordre d'exécution des threads.

2. Modifier ce programme pour avoir toujours l'affichage suivant :

main Init
thread1

thread2
main Fin
il faut ajouter avant printf("main Fin\n"):
pthread_join(t1,&status);
pthread_join(t2,&status);
avec dans la declaration : *void * status;*

Cela va permettre au processus père d'attendre la fin du thread 1 et du thread 2 c'est à dire la fin de l'exécution des fonctions exécutées par ceux-ci.

2. Modifier le dernier programme et utiliser une seule fonction task au lieu de task1 et task2. L'output doit être le même que celui de la question 2.

On va utiliser ici une fonction (task) paramétrée :

```
#include <stdio.h>
#include <pthread.h>
void *task (void *arg )
{ printf (" Thread %d \n",(int)arg );
pthread_exit (0);
return NULL;
}
int main ( )
{
pthread_t t1, t2;
int i=1 ;
int j=2 ;
void * status ;
printf ("main Init\n");
pthread_create (&t1, NULL, task,(void*)i);
pthread_create (&t2, NULL, task,(void*)j);
printf ("main Fin\n");
return 0;
}
```

Exercice 2 :

Soit le programme suivant :

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
/*1*/
int data=0;
void *fonctionplus (void *arg)
{ int i ;
for (i=0;i<4;++i)
{
/*2*/
data++;
```

```
printf("Thread plus %d\n", data);
/*3*/
}
pthread_exit(0);
return NULL;
}
void *fonctionmoins (void *arg)
{ int i;
for(i=0;i<4;++i)
{
/*4*/
data--;
printf("Thread moins %d\n", data);
/*5*/
}
pthread_exit(0);
return NULL;
}
int main (void)
{ pthread_t tid1, tid2 ;
void *status ;
int i ;
printf("main Init\n");
/*6*/
pthread_create(&tid1,NULL,fonctionplus,NULL) ;
pthread_create(&tid2,NULL,fonctionmoins,NULL);
for (i=0;i<4;++i)
{
/*7*/
printf ("\n Variable partage = %d\n",data) ;
/*8*/
}
printf ("main Fin \n") ;
pthread_join(tid1,&status) ;
pthread_join(tid2,&status);
/*9*/
return 0 ;
}
```

1. Tester ce programme

   On remarque que les deux threads ont une variable partagée (data). Pour cela, il faut les délimiter par soit les mutex soit les sémaphores.

2. Terminer les passages manquants marqués par les numéros en utilisant les mutex.

```
#include <pthread.h>
#include <unistd.h>
```

```
#include <stdio.h>
pthread_mutex_t S;
int data=0;
void *fonctionplus (void *arg)
{ int i ;
for (i=0;i<4;++i)
{
pthread_mutex_lock(&S);
data++;
printf("Thread plus %d\n", data);
pthread_mutex_unlock(&S);
}
pthread_exit(0);
return NULL;
}
void *fonctionmoins (void *arg)
{ int i;
for(i=0;i<4;++i)
{
pthread_mutex_lock(&S);
data--;
printf("Thread moins %d\n", data);
pthread_mutex_unlock(&S);
}
pthread_exit(0);
return NULL;
}
int main (void)
{ pthread_t tid1, tid2 ;
void *status ;
int i ;
printf("main Init\n");
/*6*/
pthread_create(&tid1,NULL,fonctionplus,NULL) ;
pthread_create(&tid2,NULL,fonctionmoins,NULL);
for (i=0;i<4;++i)
{
pthread_mutex_lock(&S);
printf ("\n Variable partage = %d\n",data) ;
pthread_mutex_unlock(&S);
}
printf ("main Fin \n") ;
pthread_join(tid1,&status) ;
pthread_join(tid2,&status);
```

```
pthread_mutex_destroy(&S);
return 0 ;
}
```

2. Balancer l'exécution entre le thread principal, le thread d'incrémentation et le thread de décrémentation en utilisant sleep.
   Ici on part du principe que pour alterner entre l'exécution du thread principal, le thread d'incrémentation et le thread de décrémentation, il faut que tous les threads attendent la même période.
   Aussi, nous notons que le timing est étroitement lié au processeurs.

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
pthread_mutex_t S;
int data=0;
void *fonctionplus (void *arg)
{ int i ;
for (i=0;i<4;++i)
{
sleep(1);
pthread_mutex_lock(&S);
data++;
printf("Thread plus %d\n", data);
pthread_mutex_unlock(&S);
sleep(2);
}
pthread_exit(0);
return NULL;
}
void *fonctionmoins (void *arg)
{ int i;
for(i=0;i<4;++i)
{
sleep(1);
pthread_mutex_lock(&S);
data--;
printf("Thread moins %d\n", data);
pthread_mutex_unlock(&S);
sleep(2);
}
pthread_exit(0);
return NULL;
}
int main (void)
{ pthread_t tid1, tid2 ;
```

```
void *status ;
int i ;
printf("main Init\n");
pthread_mutex_init(&S,NULL);
pthread_create(&tid1,NULL,fonctionplus,NULL) ;
pthread_create(&tid2,NULL,fonctionmoins,NULL);
for (i=0;i<4;++i)
{
pthread_mutex_lock(&S);
printf ("\n Variable partage = %d\n",data) ;
pthread_mutex_unlock(&S);
sleep(3);
}
printf ("main Fin \n") ;
pthread_join(tid1,&status) ;
pthread_join(tid2,&status);
pthread_mutex_destroy(&S);
return 0 ;
}
```

3. Reprendre le code précédent en utilisant les sémaphores.

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <semaphore.h>
sem_t S;
int data=0;
void *fonctionplus (void *arg)
{ int i ;
for (i=0;i<4;++i)
{
sleep(1);
sem_wait(&S);
data++;
printf("Thread plus %d\n", data);
sem_post(&S);
sleep(2);
}
pthread_exit(0);
return NULL;
}
void *fonctionmoins (void *arg)
{ int i;
for(i=0;i<4;++i)
{
```

```
sleep(1);
sem_wait(&S);
data--;
printf("Thread moins %d\n", data);
sem_wait(&S);
sleep(2);
}
pthread_exit(0);
return NULL;
}
int main (void)
{ pthread_t tid1, tid2 ;
void *status ;
int i ;
printf("main Init\n");
sem_init(&S,0,1);
pthread_create(&tid1,NULL,fonctionplus,NULL) ;
pthread_create(&tid2,NULL,fonctionmoins,NULL);
for (i=0;i<4;++i)
{
sem_wait(&S);
printf ("\n Variable partage = %d\n",data) ;
sem_wait(&S);
sleep(3);
}
printf ("main Fin \n") ;
pthread_join(tid1,&status) ;
pthread_join(tid2,&status);
sem_destroy(&S);
return 0 ;
}
```