Maîtrise d'informatique
Module de programmation parallèle

TD : Introduction au multithreading

Philippe Marquet

Février 2000

Exercice 1 (Compteur)
Il s'agit de développer une version « MT-Safe » de la fonction suivante de gestion d'un compteur (par exemple un compteur du nombre d'appel...).

```
int count_me (int i) {
  static int count = 0 ;        /* variable globale à visibilité locale */

  count += i ;

  return count ;
}
```

Question 1.1 Quelle problème peut survenir lors de l'utilisation de cette fonction dans un programme multithreadé ?

Question 1.2 Quelle structure utiliser pour protéger les accès à `count` ?

Question 1.3 Comment retourner la valeur de `count` ?

Exercice 2 (À la recherche du signal perdu...)
Soit l'extrait de code producteur/consommateur suivant :

```
/* Consommateur */                        /* Producteur, Signaleur */

pthread_mutex_lock (&lock) ;              expression = TRUE ;
while (!expression)                       pthread_cond_signal (&cond) ;
  pthread_cond_wait (&cond, &lock) ;

do_thing () ;
pthread_mutex_unlock (&lock) ;
```

Question 2.1 Montrer que le consommateur peut ne pas être réveillé alors qu'il le devrait : des `pthread_cond_signal()` pouvant être perdus.

Question 2.2 Quelle(s) correction(s) apporter à cet extrait de code pour éviter ce comportement ?

On reprend l'exemple du producteur/consommateur vu en cours :

```
#define QSIZE           10
typedef struct circ_buf_struct circ_buf_t ;
struct circ_buf_struct {
  pthread_mutex_t buf_lock ;    /* lock the structure */
  int start_idx ;               /* start of valid data */
  int num_full ;                /* # of full locations */
  pthread_cond_t notfull ;      /* wait on condition buffer full */
  pthread_cond_t notempty ;     /* wait on condition buffer empty*/
  void *data[QSIZE] ;           /* circular buffer of pointers */
```

Id: td-mt.tex,v 1.4 2000/02/16 14:40:33 marquet Exp

```
}

void put_cb_data (circ_buf_t *cbp, void *data)
{
  pthread_mutex_lock (&cbp->buf_lock) ;
  while (cbp-> num_full == QSIZE)
    pthread_cond_wait (&cbp->notfull, &cbp->buf_lock) ;
  cbp->data[(cbp->start_idx + cbp->num_full) % QSIZE] = data ;
  cbp->num_full++ ;
  pthread_cond_signal (&cbp->notempty) ;
  pthread_mutex_unlock (&cbp->buf_lock) ;
}

void *get_cb_data (circ_buf_t *cbp)
{
  void *data ;

  pthread_mutex_lock (&cbp->buf_lock) ;
  while (cbp-> num_full == 0)
    pthread_cond_wait (&cbp->notempty, &cbp->buf_lock) ;
  data = cbp->data[cbp->start_idx] ;
  cbp->start_idx = (cbp->start_idx + 1) % QSIZE ;
  cbp->num_full-- ;
  pthread_cond_signal (&cbp->notfull) ;
  pthread_mutex_unlock (&cbp->buf_lock) ;
  return data ;
}
```

Est-ce que dans le producteur `put_cb_data()`, on peut échanger l'ordre des instructions `pthread_cond_signal()` et `pthread_mutex_unlock()` ?

Exercice 3 (Fibonacci)

Soit le programme suivant de calcul du n^{e} nombre de Fibonacci :

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

int fib (int n)
{
  if (n<2) return n ;
  else {
    int x, y ;

    x = fib (n-1) ;
    y = fib (n-2) ;

    return x + y ;
  }
}

int main (int argc, char *argv[])
{
  int n, res ;

  n = atoi (argv[1]) ;
  res = fib (n) ;
```

```
  printf ("Fibo (%d) = %d\n", n, res) ;
  exit (0) ;
}
```

Question 3.1 Une première parallélisation à l'aide processus légers. L'idée est d'attacher un thread à chacune des instances de la fonction `fib()`. On construit ainsi un arbre de threads. Avant de retourner son résultat, un thread attend que ses deux fils aient terminé.

Question 3.2 On propose d'améliorer la solution précédente de la manière suivante : plutôt que de déclencher un thread alors que la valeur a déjà été produite auparavant par un autre thread, on récupère cette valeur. Pour cela, on garde les résultats intermédiaires produits dans un tableau partagé entre les threads. (Les éléments de ce tableau sont par exemple initialisés à la valeur `INCONNUE` pour dénoter que la valeur n'a pas encore été calculée.)

```
#define INCONNUE        -1
int fibtab [] ;
```

Question 3.3 On modifie encore la solution précédente pour marquer qu'un thread est en train de calculer une valeur. Les éléments du tableau `fibtab` ont soit une valeur positive, soit une des deux valeurs `INCONNUE` et `ENCOURS`.

```
#define INCONNUE        -1
#define ENCOURS         -2
int fibtab [] ;
```

Exercice 4 (Barrières de synchronisation)
On désire implanter une bibliothèque de barrières de synchronisation à l'aide des constructions fournies par la bibliothèques POSIX `thread` et/ou POSIX `semaphores`.

L'interface de cette bibliothèque sera la suivante :

```
typedef struct barrier_struct barrier_t ;

barrier_t *barrier_init (unsigned int nb_clients) ;
void barrier_destroy (barrier_t *barrier) ;
void barrier_wait (barrier_t *barrier ) ;
```

`barrier_init()` alloue et initialise une barrière qui sera utilisée par `nb_clients`.

`barrier_destroy()` demande la destruction de la barrière. Aucun thread ne doit attendre sur la barrière lors de la destruction.

`barrier_wait()` est appelé par un thread pour rentrer dans la barrière. Quand `nb_clients` threads ont atteint la barrière, ils sont tous libérés.

Question 4.1 Quelle structure est nécessaire à l'implantation des barrières ?

Question 4.2 Donner le code des fonctions de la bibliothèque.

On considère maintenant l'utilisation suivante de la bibliothèque :

```
#define N               ...
barrier_t b ;

void foo (void)
{
  do_thing1 () ;
  barrier_wait (&b) ;

  do_thing2 () ;
```

```
  barrier_wait (&b) ;

  pthread_exit (0) ;
}

int main (void)
{
  int i ;
  pthread_t dummy_tid ;
  b = barrier_init (N) ;

  for (i=1 ; i<N ; i++)
    pthread_create (&dummy_tid, NULL, (void *) foo, (void *) NULL) ;

  foo () ;
}
```

Question 4.3 Considérer les différents ordonnancements des threads pouvant intervenir entre la première et la seconde utilisation de la barrière. Mettre en évidence un fonctionnement erroné possible.

Question 4.4 Corriger la première implantation de la bibliothèque pour que l'utilisation multiple des barrières ne pose aucun problème.

Exercice 5 (Variante : barrières cumulatives)
On change l'interface de la bibliothèque. La fonction `barrier_wait()` devient

```
int barrier_wait (barrier_t *barrier, int increment) ;
```

Elle est appelé par un thread qui atteint la barrière. Quand `nb_clients` threads ont atteint la barrière, ils sont tous libérés. Le résultat retourné est la somme de tous les `increment` passés par les threads lors de leur appel à `barrier_wait()`.

Exercice 6 (Variante : barrières asymétriques)
On change encore l'interface de la bibliothèque. La fonction `barrier_wait()` est remplacée par trois nouvelles fonctions. L'utilisation des deux fonctions `barrier_init()` et `barrier_destroy()` change légèrement :

```
barrier_t *barrier_init (unsigned int n_slaves) ;
void barrier_destroy (barrier_t *barrier) ;
int barrier_master (barrier_t *barrier) ;
int barrier_release (barrier_t *barrier) ;
int barrier_slave (barrier_t *barrier, int increment) ;
```

`barrier_init()` est appelé par le thread maître pour allouer et initialiser une barrière qui sera utilisée avec `n_slaves` threads esclaves.

`barrer_destroy()` est appelé par le thread maître pour détruire la barrière.

`barrier_master()` est appelé par le thread maître pour attendre que les `n_slaves` threads esclaves aient atteint la barrière. Quand c'est le cas, `barrier_master()` retourne la somme des incréments donnés par l'ensemble des esclaves.

`barrier_release()` est appelé par le maître pour libérer l'ensemble des esclaves qui attendent sur la barrière.

`barrier_slave()` est appelé par un thread esclave qui atteint la barrière. La valeur `increment` est accumulée pour être retournée (`barrier_master()`) au maître. `barrier_slave()` retourne après que le maître ait appelé `barrier_release()`. La valeur retournée est la somme des `increment` de l'ensemble des esclaves (i.e. la même valeur que celle retournée au maître par `barrier_master()`).

Question 6.1 Quelle utilité voyez vous à l'utilisation d'une telle structure de synchronisation ?

Question 6.2 Donner une implantation de la bibliothèque `barrier` initiale à l'aide de cette version asymétrique.

Question 6.3 Donner une implantation de cette version asymétrique à l'aide de la version initiale de la bibliothèque.

Question 6.4 Donner une implantation de cette version de la bibliothèque.

Exercice 7 (World Hello)
Soit le programme `hello_world.c` suivant.

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

/* Un thread fils affiche un message */
void print_message (void *arg) ;

/* Terminaison sur une erreur. */
void die (int status, char *message) ;

int main (void)
{
  pthread_t tid1, tid2 ;        /* les deux fils */
  char *message1 = "Hello" ;    /* les messages des fils */
  char *message2 = "World" ;
  int status ;

  /* lancement du premier fils */
  status = pthread_create (&tid1, NULL,
                           (void *) print_message , (void *) message1) ;
  if (status) die (status, "Creation du premier fils") ;

  /* Lancement du second */
  status = pthread_create (&tid2, NULL,
                           (void *) print_message , (void *) message2) ;
  if (status) die (status, "Creation du second fils") ;

  exit (0) ;
}

void print_message (void *arg)
{
  char *mess = (char *) arg ;
  printf ("%s ", mess) ;
  fflush (stdout) ;

  pthread_exit (0) ;
}
```

Question 7.1 Donner les différents résultats pouvant être produits par ce programme.

Question 7.2 Corriger le programme pour qu'il affiche `"Hello World"`.

Exercice 8 (Verrous récursifs)
On désire implanter une extension des verrous `mutex` de la bibilothèque `pthread` de base. Un de nos `rec_mutex` peut à nouveau être `rec_mutex_lock()` par le thread propriétaire du verrou sans qu'il se bloque. Le proprietaire d'un verrou le relâche par un nombre d'appels à `rec_mutex_unlock()` égal au nombre d'appels à `rec_mutex_lock()` réalisés.

Question 8.1 Définir un type et écrire les fonctions suivantes :

```
typedef struct _rec_mutex_t rec_mutex_t ;

struct _rec_mutex_t { ... } ;

#define REC_MUTEX_INITIALIZER ...

int rec_mutex_init (rec_mutex_t *rm) ;
int rec_mutex_destroy (rec_mutex_t *rm) ;

int rec_mutex_lock (rec_mutex_t *rm);
int rec_mutex_unlock (rec_mutex_t *mutex);
```

Question 8.2 Définir la fonction :

```
int rec_mutex_trylock (rec_mutex_t *rm) ;
```

Exercice 9 (Conditions FIFO)
Les conditions disponibles dans les implantations de la norme POSIX `pthread` n'ont pas à garantir d'ordre sur le réveil des threads bloqués sur une condition. Nous désirons implanter une bibliothèque `fifo_cond` de conditions FIFO. Ce type de bibliothèque est utile si on veut implanter des algorithmes d'exclusion mutuelle garantissant certaines priorités (par exemple les lecteurs/rédacteurs avec priorité égale ; cf les exercices sur les moniteurs de Hoare à ce sujet).

Exercice 10 (Verrou FIFO)
On suppose avoir à notre disposition une bibliothèque de conditions FIFO (voir exercice 9). Nous désirons proposer une bibliothèque `fifo_mutex` de verrous FIFO.