INF8541: Devoir #4

(À remettre le 23 octobre)

1. Mise en oeuvre de I-structures avec sémaphores

Une I-structure est une forme de structure de données utilisée pour la programmation parallèle qui permet la synchronisation *implicite* entre producteur et consommateur(s). Plus précisément, les I-structures supportent des opérations ayant la forme suivante:

- `void I_ALLOC( int nb, I_STRUCTURE *i_s )`: Alloue une I-structure (un tableau) pouvant contenir `nb` éléments et retourne cette I-structure dans `*i_s`. Initialement, chacune des cellules de la I-structure est *vide*, c'est-à-dire ne contient aucune valeur.

- `void I_WRITE( I_STRUCTURE i_s, int index, T val, bool *ok )`: Écrit la valeur `val` à la position indiquée par `index`. Si cette position a déjà été remplie, une erreur est signalée (`*ok == FALSE`, sinon `*ok == TRUE`). Si des requêtes de lecture sont en attente pour cet index, alors elles sont réactivées.

- `void I_READ( I_STRUCTURE i_s, int index, T *val_lue )`: Retourne dans `val_lue` le contenu de la I-structure à l'`index` indiqué. Si la cellule associée est vide, alors la requête est suspendue jusqu'à ce qu'une écriture appropriée soit effectuée.

Modifiez et complétez, dans la notation d'Andrews et *en utilisant des sémaphores*, la mise en oeuvre des I-structures présentée plus bas. Assurez-vous que le comportement sera correct en présence de *threads* concurrents. Vous pouvez indiquer vos modifications et ajouts directement sur le code existant. Vous pouvez aussi obtenir ce code à l'URL suivant:

`http://www.info.uqam.ca/~tremblay/INF8541/Devoirs`

```
typedef struct {
  bool vide;          # Indique si la cellule est vide ou non.
  T contenu;          # Valeur conservee par la cellule (si pas vide).
  # A completer
  ...
} I_struct_item;

typedef I_struct_item* I_STRUCTURE;

void I_ALLOC( int nb, I_STRUCTURE* i_s )
{
  int i;

  MALLOC_N( *i_s, I_struct_item, nb );
  for[i = 0 to nb-1] {
    (*i_s)[i].vide = TRUE;



  }
}

void I_READ( I_STRUCTURE i_s, int index, T *val_lue )
{
  # A completer.
  ...




}

void I_WRITE( I_STRUCTURE i_s, int index, T val, bool *ok )
{

  if ( i_s[index].vide ) {
    *ok = TRUE;
    i_s[index].contenu = val;
    i_s[index].vide    = FALSE;
    # Reactiver chacun des lecteurs en attente.
    ...
  } else {
    *ok = FALSE;     # Erreur: contenu deja defini.
  }

}
```

2. Pthreads

En utilisant les opérations de la bibliothèque *Pthreads*, développez le code permettant de définir la fonction `fact` qui calcule la factorielle d'un nombre à l'aide de l'approche récursive suivante:

```
int fact( int inf, int sup, int coupure )
{
   if( sup-inf+1 <= coupure ) {
     int res = 1;
     for[i = inf to sup]
       res *= i;
     return( res );
   } else {
     int res1, res2;
     co res1 = fact( inf,          (inf+sup)/2, coupure );
     // res2 = fact( (inf+sup)/2+1, sup,        coupure );
     oc
     return( res1 * res2 );
   }
 }

 Appel:
 -----
   res = fact( 1, n, coupure );
```

Le programme principal qui pourrait appeler votre procédure est présenté plus bas. Le code pour ce programme principal est aussi disponible à l'URL suivant:

`http://www.info.uqam.ca/~tremblay/INF8541/Devoirs`

Note: Les opérations `pthread_exit` et `pthread_join`, d'après leur interface, doivent manipuler des valeurs de type `void*`. Toutefois, il est possible d'utiliser sans problème un `int` dans le contexte de `pthread_exit`, évidemment en indiquant une opération de coercion de type (*type cast*) appropriée.

```
/* Factoriel: version recursive parallele dichotomique avec valeur de coupure.
   Compilation et execution sur Solaris:
     gcc THISFILE.c -o fact -lpthread -lposix4
     fact n coupure
*/

#include <pthread.h>
#include <stdio.h>
#include <assert.h>

/* Attribut (global) pour creation des divers threads. */
#define SHARED 1
pthread_attr_t attr;

/* Structure pour les arguments lors des appels a fact. */
typedef struct {
  int inf, sup;
  int coupure;
} Args;

void* fact(void* _args);

int main(int argc, char *argv[]) {
  pthread_t fid;
  int n,        /* Valeur lue a l'appel du programme. */
      res,      /* Variable ou mettre le resultat. */
      coupure;  /* Valeur de coupure pour la recursion. */
  Args args;    /* Les arguments. */

  /* Initialisation (standard) de la structure d'attribut pour le thread. */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* Lecture des arguments a utiliser pour l'appel. */
  assert( argc >= 3 );
  n = atoi(argv[1]);
  coupure = atoi(argv[2]);
  assert( n >= 0 && coupure >= 1 );

  /* Creation du thread principal. */
  args.inf = 1;
  args.sup = n;
  args.coupure = coupure;
  pthread_create(&fid, &attr, fact, &args);

  /* Attente pour le thread enfant se termine. */
  pthread_join(fid, (void *) &res);
  printf( "fact( %d ) = %d\n", n, res );
}

/* Definition de la procedure fact. */
void* fact(void* _args)
{
  # Procedure a completer pour l'exercice.
  ...
}
```