Parallélisme et concurrence avec les *threads* Posix

1 Extraits du fichier `count.c`

```
// Version avec parallelisme recursif, mais qui cree un seul nouveau
// thread, pour un seul des sous-problemes, l'autre etant traite de
// facon sequentielle (par le thread parent).

// Structure pour les arguments lors des appels a count.
typedef struct {
  int* a;         // Le tableau a analyser.
  int inf, sup;   // Les bornes de la tranche a traiter.
  int seuil;      // Le seuil de recursion.
  Predicat pred;
  int en_parallele; // Si executee de facon parallele (1), sinon (0).
} ArgsCount1;

//
```

```
void* threaded_count1( void* _args ) {
  ArgsCount1 *args;
  args = (ArgsCount1 *) _args; // Pour eviter les casts subsequents.
  if (args->sup - args->inf + 1  <= args->seuil) {
    // Cas de base.
    long nb = count_seq( args->a, args->inf, args->sup, args->pred );
    if (args->en_parallele)
      pthread_exit( (void*) nb );
    else
      return (void*) nb;
  }

  // Cas recursif.
  ArgsCount1 args1, args2;
  args1 = args2 = *args;
  int mid = (args->inf + args->sup) / 2;
  args1.sup = mid;
  args2.inf = mid + 1;

  // On cree le thread pour qu'il execute le premier appel en parallele.
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_t thread1;
  args1.en_parallele = 1;
  pthread_create(&thread1, &attr, threaded_count1, (void*) &args1);

  // On traite le deuxieme appel de facon sequentielle.
  args2.en_parallele = 0;
  long nb2 = (long) threaded_count1( (void*) &args2 );

  // On attend le resultat du premier thread.
  long nb1;
  pthread_join(thread1, (void*) &nb1);

  // On combine les solutions et on retourne le resultat final.
  if (args->en_parallele)
    pthread_exit( (void*) nb1 + nb2 );
  else
    return (void *) nb1 + nb2;
}
```

```
// Fonction principale, qui effectue le premier appel, mais sans
// necessairement lancer un nouveau thread.
long count_rec_par1( int a[], int inf, int sup, int seuil, Predicat pred ) {
  // On definit les arguments pour le premier appel, qui sera execute
  // par le thread courant.
  ArgsCount1 args;
  args.a = a;
  args.inf = inf;
  args.sup = sup;
  args.seuil = seuil;
  args.pred = pred;
  args.en_parallele = 0;

  // On lance l'appel.
  return (long) threaded_count1( &args );
}
```

2 Extraits du fichier svar.c

```
// Constructeur de base.
SVar new_SVar()
{
  SVar sv = (SVar) malloc( sizeof(SVarStruct) );
  sv->state = EMPTY;

  pthread_mutex_init(&sv->mutex, NULL);
  pthread_cond_init(&sv->full, NULL);

  return sv;
}

// Mutateur.
void set_value( SVar sv, Value value )
{
  pthread_mutex_lock(&sv->mutex);
  assert( is_empty(sv) && "*** sv pas vide!?" );

  sv->value = value;
  sv->state = FULL;
  pthread_cond_broadcast(&sv->full);
  pthread_mutex_unlock(&sv->mutex);
}

// Observateurs.
Value value( SVar sv )
{
  pthread_mutex_lock(&sv->mutex);
  if ( is_empty(sv) ) {
    pthread_cond_wait(&sv->full, &sv->mutex);
  }
  pthread_mutex_unlock(&sv->mutex);

  return sv->value;
}
//
```

```
typedef struct {
  SVar svToWait;
  SVar svResult;
  Value (*proc)(Value);
} ArgsThen;

static Value threaded_then( Value args ) {
  ArgsThen* args_then = (ArgsThen*) args;

  Value val = value(args_then->sv_to_wait);
  Value res = args_then->proc(val);
  set_value( args_then->sv_result, res );

  pthread_exit( NULL );
}

SVar then( SVar sv, Value (*proc)(Value)) {
  SVar res = new_SVar();
  ArgsThen* args = (ArgsThen*) malloc( sizeof(ArgsThen) );
  args->sv_to_wait = sv;
  args->sv_result = res;
  args->proc = proc;

  pthread_attr_t attr;
  pthread_t thread;
  pthread_attr_init(&attr);

  pthread_create(&thread, &attr, threaded_then, (void*) args );

  return res;
}
```