Table des matières

Chapitre 12

Programmation parallèle et concurrente en C avec les *threads* Posix

Ce chapitre présente les notions de base pour l'utilisation des *threads* Posix en C — bibliothèque `pthread` — ainsi que quelques exemples d'utilisations.

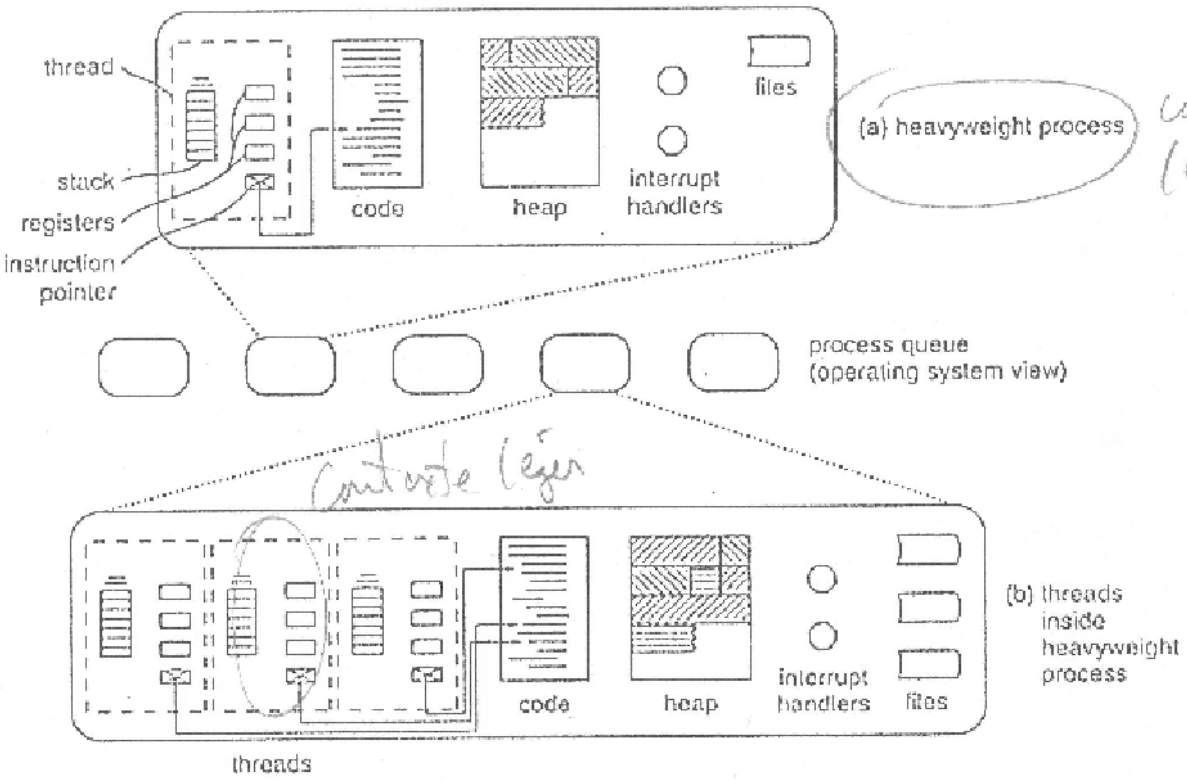

Figure 4.1
Process Organization

Figure 12.0: Processus vs. *thread* dans l'environnement Unix (source inconnue).

12.1 Historique

Les mécanismes de création de processus de style Unix (`fork`, `wait`, `pipe`) sont très coûteux et ne peuvent être utilisés que pour des programmes avec un nombre *restreint* de processus — avec un petit nombre de processus *poids lourd*.

Au milieu des années 90, diverses *bibliothèques* de *threads* **poids légers** furent introduites, souvent associées au langage C. Une des plus connues, maintenant devenue un standard, est la bibliothèque de *threads* Posix = *Portable Operating System Interface* — d'où le nom `pthread`.

La bibliothèque `pthread` définit des douzaines de fonctions, dans le cadre du langage C, pour la programmation concurrente avec *thread*s.

Comme on le verra, le modèle de programmation est uniquement **du parallélisme *fork/join*!**

12.2 Création de *threads*

Les principales opérations associées à la création de *threads* sont les suivantes :

- Inclusion de la bibliothèque :

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

- Déclaration des attributs décrivant les *threads* :

```
/* Attributs du thread. */
pthread_attr_t tattr;

/* Descripteur (identifiant) du thread. */
pthread_t tid;
```

- Initialisation des attributs des *threads* :

```
/* Utilisation des attributs par défaut. */
pthread_attr_init( &tattr );
```

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);


Compile and link with -pthread.
```

DESCRIPTION

The `pthread_attr_init()` *function initializes the thread attributes object pointed to by* `attr` **with default attribute values**. *After this call, individual attributes of the object can be set using various related functions, and then the object can be used in one or more* `pthread_create` *calls that create threads.*

Compléments d'information 1: Quelques détails additionnels sur la fonction `pthread_attr_init` et les attributs associés.

Quelques fonctions pour spécifier des attributs autres que les valeurs par défaut :

- `pthread_attr_setaffinity_np`
- `pthread_attr_setdetachstate`
- `pthread_attr_setguardsize`
- `pthread_attr_setinheritsched`
- `pthread_attr_setschedparam`
- `pthread_attr_setschedpolicy` : «*This attribute determines the scheduling policy of a thread created using the thread attributes object attr. The supported values for policy are* `SCHED_FIFO`, `SCHED_RR`, *and* `SCHED_OTHER`*[.]*»
- `pthread_attr_setscope` : «*sets the contention scope attribute of the thread attributes object referred to by* `attr` *to the value specified in* `scope`*. The contention scope attribute defines the set of threads against which a thread competes for resources such as the CPU.*»

  Note : *POSIX.1 requires that an implementation support at least one of these contention scopes. Linux supports* `PTHREAD_SCOPE_SYSTEM`*, but not* `PTHREAD_SCOPE_PROCESS`*.*
- `pthread_attr_setstack` : «*These attributes specify the location and size of the stack that should be used by a thread that is created using the thread attributes object* `attr`.»
- `pthread_attr_setstackaddr`
- `pthread_attr_setstacksize`
- `pthread_setattr_default_np`

Compléments d'information 1: Quelques détails additionnels sur la fonction `pthread_attr_init` et les attributs associés.

- Création et activation du *thread* par l'appel d'une fonction `func` avec un argument `arg` :

  ```
  pthread_create( &tid, &tattr, func, arg );
  ```

  La fonction `func` associée au *thread doit nécessairement avoir la signature suivante* :

  ```
  void* func( void* )
  ```

  Un seul argument peut être transmis à l'appel. Par contre, cet argument peut évidemment être une structure complexe ayant plusieurs éléments : voir le programme présenté à la section 12.6.

  L'opération `pthread_create` retourne 0 si le *thread* a été créé correctement ; dans ce cas, `tid` contient l'identifiant (le descripteur) du *thread* créé.

- Attente pour qu'un *thread* se termine — un parent (créateur d'un *thread*) peut attendre qu'un enfant se termine :

  ```
  pthread_join( tid, &result );
  ```

  La valeur dans `result` (sauf si on a indiqué `NULL`) est celle spécifiée par le *thread* lorsqu'il se termine avec :

  ```
  pthread_exit( return_value );
  ```

  Note : Si la fonction pour un *thread* se termine sans un appel explicite à `pthread_exit`, alors `NULL` est retournée. On peut aussi retourner `NULL` de façon explicite si aucun résultat n'a à être produit.

12.3 Sémaphores

Les *thread* Posix communiquent entre eux par l'intermédiaire de variables partagées déclarées de façon globale (à l'extérieur des fonctions associées aux *thread*s). Les *threads* peuvent se synchroniser entre eux par l'intermédiaire de *sémaphores*, verrous ou variables de condition.

Utilisation de sémaphores :

- Inclusion du fichier approprié :

  ```
  #include <semaphore.h>
  ```

- Déclaration d'une variable globale de type `sem_t` :

  ```
  sem_t mutex;
  ```

- Initialisation

  ```
  sem_init( &mutex, 0, 1 );
  ```

  Le deuxième argument (ici, 0) indique si le sémaphore est partagé entre les *thread*s d'un même processus (0) — on indique la valeur 1 pour qu'il soit partagé entre les *thread*s de processus indépendants (1).

  Le troisième argument indique la valeur initiale du sémaphore. Cette valeur sera 1 dans le cas d'un sémaphore utilisé comme verrou.

- Les opérations de base :

  ```
  sem_wait( &mutex );   /* P(mutex); */
  sem_post( &mutex );   /* V(mutex); */
  ```

De nombreuses autres opérations sont disponibles : attente conditionnelle, examen de la valeur courante du sémaphore, destruction d'un sémaphore.

NAME

semaphore.h - semaphores (**REALTIME**)

SYNOPSIS

[SEM] ⌦ #include <semaphore.h> ⌫

DESCRIPTION

The <*semaphore.h*> header shall define the **sem_t** type, used in performing semaphore operations. The semaphore may be implemented using a file descriptor, in which case applications are able to open up at least a total of {OPEN_MAX} files and semaphores. The symbol SEM_FAILED shall be defined (see *sem_open()*).

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int     sem_close(sem_t *);
int     sem_destroy(sem_t *);
int     sem_getvalue(sem_t *restrict, int *restrict);
int     sem_init(sem_t *, int, unsigned);
sem_t *sem_open(const char *, int, ...);
int     sem_post(sem_t *);
[TMO]⌦
int     sem_timedwait(sem_t *restrict, const struct timespec *restrict);
⌫
int     sem_trywait(sem_t *);
int     sem_unlink(const char *);
int     sem_wait(sem_t *);
```

Inclusion of the <*semaphore.h*> header may make visible symbols defined in the headers <*fcntl.h*> and <*sys/types.h*>.

Figure 12.1: Spécification du type `sem_t`.

Les sémaphores ont plus de fonctionnalités que de simples verrous :

Programme Java 12.1 Moniteur Java pour une classe `Semaphore` simplifiée ≈ semblable au type `sem_t` de Posix.

```
public class Semaphore {
  private int val;

  public Semaphore( int valInitiale ) {
    assert valInitiale >= 0;
    val = valInitiale;
  }

  public synchronized int wait() {
    while ( val == 0 ) {
      try { wait(); } catch ( Exception e ) {...}
    }
    val = val - 1;

    return 0; // Success!
  }

  public synchronized int post() {
    val = val + 1;
    notify();

    return 0; // Success!
  }
}
```

12.4 Verrous et variables de condition

- Déclaration et initialisation d'un verrou ou d'une variable de condition :

```
pthread_mutex_t mutex;
pthread_mutex_init( &mutex, NULL );

pthread_cond_t  cond;
pthread_cond_init ( &cond,  NULL );
```

Note : Lors de l'initialisation, on peut aussi spécifier divers attributs, par ex., gestion de l'inversion de priorité, traitement des verrouillages récursifs (réentrants ou non), portée de la variable (système, processus), etc.

Note : *Passing* `NULL` *is equivalent to passing a mutex attribute object with all attributes set* to their default values. *[. . .] The default mutex type is* `PTHREAD_MUTEX_NORMAL`.

- Accès à une section critique :

```
pthread_mutex_lock( &mutex );
...
pthread_mutex_unlock( &mutex );
```

- Attente sur une condition — **le *thread* doit avoir le verrou en sa possession** :

```
pthread_cond_wait( &cond, &mutex );
```

- Signal d'une variable de condition (signaler et continuer) :

```
pthread_cond_signal( &cond );

pthread_cond_broadcast( &cond );
```

PTHREAD_COND

Section: C Library Functions (3)
Updated: LinuxThreads
Index Return to Main Contents

NAME

pthread_cond_init, pthread_cond_destroy, pthread_cond_signal, pthread_cond_broadcast, pthread_cond_wait, pthread_cond_timedwait - operations on conditions

SYNOPSIS

#include <pthread.h>

pthread_cond_t *cond* **= PTHREAD_COND_INITIALIZER;**

int pthread_cond_init(pthread_cond_t ***cond*****, pthread_condattr_t** ****cond_attr*****);**

int pthread_cond_signal(pthread_cond_t ***cond*****);**

int pthread_cond_broadcast(pthread_cond_t ***cond*****);**

int pthread_cond_wait(pthread_cond_t ***cond*****, pthread_mutex_t** ****mutex*****);**

int pthread_cond_timedwait(pthread_cond_t ***cond*****, pthread_mutex_t** ****mutex*****, const struct timespec** ****abstime*****);**

int pthread_cond_destroy(pthread_cond_t ***cond*****);**

Figure 12.2: Spécification du type `pthread_cond_t`.

12.5 Exemple : Calcul de π par une méthode Monte Carlo

Le programme C qui suit, qui utilise des *threads* Posix, définit une fonction pour calculer π par une méthode Monte Carlo — donc tel que vu précédemment en Ruby.

```
//////////////////////////////
// FICHIER pi.c
//////////////////////////////

/* Programme pour approximer la valeur de pi
   a l'aide d'une methode Monte Carlo.

   gcc pi.c -o pi -std=c99 -lpthread
   pi nbLancers nbThreads
*/


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


//
```

```
void* nbDansCercleSeq( void* nbLancers )
{
  // On effectue les lancers.
  long nb = 0;
  for( long i = 0; i < (long) nbLancers; i++ ) {
    double x = (double) rand() / RAND_MAX;
    double y = (double) rand() / RAND_MAX;
    if ( x * x + y * y <= 1.0 ) {
      nb += 1;
    }
  }

  // On retourne le resultat au parent.
  pthread_exit( (void*) nb );
}

//
```

```
double evaluerPi( long nbLancers, long nbThreads )
{
  // On initialise la structure
  // d'attribut pour le thread (standard).
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );

  // On cree les threads.
  pthread_t threads[nbThreads];
  for ( long i = 0; i < nbThreads; i++ ) {
    // On passe l'unique argument entier
    // par l'intermediaire d'un cast.
    pthread_create( &threads[i],
                    &attr,
                    nbDansCercleSeq,
                    (void *) (nbLancers / nbThreads) );
  }

  // On obtient les resultats des threads.
  long nbTotalDansCercle = 0;
  for ( long i = 0; i < nbThreads; i++ ) {
    long res;
    pthread_join( threads[i], (void *) &res );
    nbTotalDansCercle += res;
  }

  return 4.0 * nbTotalDansCercle / nbLancers;
}

//
```

```
// Programme principal
int main( int argc, char *argv[] )
{
  assert( argc >= 3 );

  long nbLancers = atoi(argv[1]);
  assert( nbLancers > 0 );

  long nbThreads = atoi(argv[2]);
  assert( nbThreads > 0 );

  double pi = evaluerPi( nbLancers, nbThreads );

  printf( "pi = %f\n", pi );

  return 0;
}
```

12.6 Exemple : Somme de deux tableaux

Le programme C qui suit, qui utilise des *threads* Posix, définit une fonction pour effetuer la somme de deux tableaux — donc tel que vu précédemment en Ruby.

```
////////////////////////////////
// FICHIER somme-tableaux.c
////////////////////////////////

/* Programme qui genere deux tableaux puis qui calcule
   leur somme, en parallele.

   gcc somme-tableaux.c -o somme-tableaux -std=c99 -lpthread
   somme-tableaux nbElements nbThreads
*/

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

// Structure pour le transfert des arguments aux threads.
typedef struct {
  double *a;
  double *b;
  double *c;
  long nbElements;
} Args;

//
```

```
// La fonction executee par les threads.
void* somme_tableaux_seq( void* arg )
{
  // On identifie les "vrais" arguments.
  Args *args = (Args *) arg; // Evite les cast subsequents.
  double *a = args->a;
  double *b = args->b;
  double *c = args->c;
  long nbElements = args->nbElements;

  // On fait la somme des elements du bloc (de la tranche).
  for( long i = 0; i < nbElements; i++ ) {
    c[i] = a[i] + b[i];
  }

  // Aucun resultat a retourner.
  pthread_exit( NULL );
}


//
```

```
// La fonction principale.
void somme_tableaux_par( double a[], double b[], double c[],
                         long nbElements, long nbThreads )
{
  // On initialise les attributs des threads.
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );

  // On lance les threads: distribution par blocs.
  pthread_t threads[nbThreads];
  Args args[nbThreads];
  long nbParThread = nbElements / nbThreads;
  for ( long i = 0; i < nbThreads; i++ ) {
    // On passe directement l'adresse de debut du bloc a traiter.
    args[i].a = a + i * nbParThread;
    args[i].b = b + i * nbParThread;
    args[i].c = c + i * nbParThread;
    args[i].nbElements = nbParThread;
    pthread_create( &threads[i],
                    &attr,
                    somme_tableaux_seq,
                    &args[i] );
  }

  // On attend que les threads se terminent.
  for ( int i = 0; i < nbThreads; i++ ) {
    pthread_join( threads[i], NULL );
  }
}
//
```

```
// Le programme principal.
int main( int argc, char *argv[] )
{
  // On lit les arguments.
  assert( argc >= 3 );

  long nbElements = atoi(argv[1]);
  assert( nbElements > 0 );

  long nbThreads = atoi(argv[2]);
  assert( nbThreads > 0 );

  // On s'assure que chaque thread aura le meme nombre d'elements a traiter.
  assert( nbElements % nbThreads == 0 );

  // On alloue et inialise les tableaux.
  double *a = (double*) malloc( nbElements * sizeof(double) );
  double *b = (double*) malloc( nbElements * sizeof(double) );
  double *c = (double*) malloc( nbElements * sizeof(double) );
  for( long i = 0; i < nbElements; i++ ) { a[i] = 1.0; b[i] = 10.0; }

  // On effectue la somme, en parallele.
  somme_tableaux_par( a, b, c, nbElements, nbThreads );

  // On verifie le resultat.
  for( long i = 0; i < nbElements; i++ ) {
    assert( c[i] == (a[i] + b[i]) );
  }

  return 0;
}
```

12.7 Exemple : Somme des éléments d'un matrice

Le programme C qui suit, qui utilise des *threads* Posix, effectue la somme des éléments d'un matrice.

Ce programme est un exemple tiré du livre d'Andrews sur le langage MPD [And00] : il a été légèrement modifié pour compiler et s'exécuter correctement sur Mac OS X. Il est surtout intéressant parce qu'il définit une *barrière de synchronisation*, réalisée comme moniteur avec verrou et variable de condition.

- Le *thread* du programme principal définit la matrice à traiter. La matrice est *bidon* — elle est générée en mettant simplement des 1 partout.
- Le *thread* du programme principal active les *threads* travailleurs puis attend qu'ils se terminent.
- Chaque travailleur traite une tranche du tableau (i.e., fait la somme pour la tranche associée) et met son résultat intermédiaire dans une case du tableau `sums`.
- Lorsque tous les *threads* ont effectué leur somme (ce qui est détecté par une barrière de synchronisation), le *thread* 0 cumule alors les résultats intermédiaires puis imprime le résultat global.

```
////////////////////////////
// FICHIER matrix-sum.c
////////////////////////////

/* matrix summation using pthreads

   usage on Solaris:
     gcc matrix.sum.c -lpthread -lposix4
     a.out size numWorkers

*/

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

#define SHARED 1
#define MAXSIZE 10000  /* maximum matrix size */
#define MAXWORKERS 4   /* maximum number of workers */

//
```

```
pthread_mutex_t barrier;  /* mutex lock for the barrier */
pthread_cond_t go;        /* condition variable for leaving */

int numWorkers;
int numArrived = 0;       /* number who have arrived */

/* a reusable counter barrier */
void Barrier() {
  pthread_mutex_lock(&barrier);

  numArrived++;
  if (numArrived == numWorkers) {
    numArrived = 0;
    pthread_cond_broadcast(&go);
  } else {
    pthread_cond_wait(&go, &barrier);
  }

  pthread_mutex_unlock(&barrier);
}

//
```

```
int size, stripSize;  /* assume size is multiple of numWorkers */
int sums[MAXWORKERS];
int matrix[MAXSIZE][MAXSIZE];
void *Worker(void *);

//
```

```
/* read command line, initialize, and create threads */
int main(int argc, char *argv[]) {
  long i, j;
  pthread_attr_t attr;
  pthread_t workerid[MAXWORKERS];

  /* set global thread attributes */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  /* initialize mutex and condition variable */
  pthread_mutex_init(&barrier, NULL);
  pthread_cond_init(&go, NULL);

  /* read command line */
  size = atoi(argv[1]);
  numWorkers = atoi(argv[2]);
  stripSize = size/numWorkers;

  /* initialize the matrix */
  for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
      matrix[i][j] = 1;
    }
  }

  //
```

```
  /* create the workers, then wait */
  for (i = 0; i < numWorkers; i++) {
    pthread_create(&workerid[i], &attr, Worker, (void *) i);
  }
  for ( long i = 0; i < numWorkers; i++ ) {
    pthread_join( workerid[i], NULL );
  }
  pthread_exit(NULL);
}

//
```

```
/* Each worker sums the values in one strip of the matrix.
   After a barrier, worker(0) computes and prints the total */
void *Worker(void *arg) {
  long myid = (long) arg;
  long total, i, j, first, last;
  printf("worker %ld (pthread id %ld) has started\n",
         myid, (long) pthread_self());

  /* determine first and last rows of my strip */
  first = myid*stripSize;
  last = first + stripSize - 1;

  /* sum values in my strip */
  total = 0;
  for (i = first; i <= last; i++) {
    for (j = 0; j < size; j++) {
      total += matrix[i][j];
    }
  }
  sums[myid] = total;
  Barrier();
  if (myid == 0) {
    total = 0;
    for (i = 0; i < numWorkers; i++) {
      total += sums[i];
    }
    printf("the total is %ld\n", total);
  }

  return NULL;
}
```

Références

[And00] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, Reading, MA, 2000. [QA76.58A57 2000].