Activités dans les processus (threads)

Hinde Bouziane (bouziane@lirmm.fr)

Exemple conducteur (1/2)

- On veut traiter une image, c'est-à -dire une matrice de points (pixels). Chaque point est une structure de données (couleur, profondeur, etc).
- Le traitement d'un pixel est indépendant du traitement d'un autre pixel.
- L'idée est donc de permettre l'exécution parallèle de plusieurs traitements.
- On veut en particulier faire un traitement en parallèle sur les lignes impaires et les lignes paires (degrés de parallélisme = 2).
- On définit deux fonctions *IImpair()* (pour le traitement des lignes impaires) et *IPair()* (resp. lignes paires).

Exemple conducteur (2/2)

- **Question 1** : Pourquoi paralléliser ?
- **Question 2** : Le compilateur (ou système) est-il capable de détecter des appels de fonctions indépendantes ?
- **Question 3** : Avec les moyens que vous connaissez, proposez une solution.

Retour sur la notion de processus

- Un processus constitue une seule **unité d'exécution** qui s'exécute séquentiellement sur un seul processeur
- Un moyen pour faire du parallélisme consiste à créer plusieurs processus. Dans ce cas :
  - le changement de contexte peut être coûteux
  - l'espace d'adressage du processus n'est pas partageable : obligation de passer par des outils de communication (tubes, files de messages, mémoires partagées et autres outils)
  - outils de synchronisation entre processus difficiles.

Nous allons voir ...

- Comment paralléliser au sein d'un même processus
- Partager des données entre différentes parties parallèles
- Comment gérer la synchronisation (exclusion mutuelle, imposer un ordre, etc)

Notion de *thread*

Thread = fil en anglais. Un thread est un fil d'exécution

- Traductions : activité, tâche, fil d'exécution ou processus léger
- Les threads permettent de dérouler plusieurs suites d'instructions, en parallèle, à l'intérieur d'un même processus
- Concrètement, un thread exécute une fonction avec les propriétés suivantes :
  - chaque thread a sa propre pile et des variables locales
  - un thread peut partager des données en mémoire avec d'autres threads : la communication entre threads se fait via le partage de variables.
  - la fonction s'exécute de façon *asynchrone*.

Notion de *thread* - suite

- Dans un processus, on aura un thread *principal*, celui exécutant la fonction `main()` et des threads *secondaires*.
- Sur une machine multi-processeur, chaque thread peut s'exécuter sur un processeur, indépendamment d'un autre.
- Le système gère la commutation de contexte entre threads.

Dans ce cours, l'interface de programmation normalisée POSIX est utilisée. Le mot *pthread* désignera les threads tels qu'ils sont vus dans cette interface.

Retour à l'exemple conducteur

Rappel : deux fonctions, *lImpair()* et *lPair()*, pouvant s'exécuter en parallèle.

Un schéma possible de traitement de l'image avec des threads :

```
int main(){
   ...
   // définition de la matrice (image)
   ...
   pthread_create(lImpair,...);
   pthread_create(lpair,...);
   ...
}
```

Création

Prototype :

```
int    pthread_create(                  // résultat 0 si réussite, ≠ 0 sinon
       pthread_t * idThread,            // identité obtenue en résultat
       pthread_attr_t *attributs,       // NULL pour commencer
       void * (*fonction)(void *),      // fonction à démarrer
       void * param) ;                  // paramètre(s) à passer à la fonction
```

Cette fonction démarre l'exécution d'un nouveau thread, en parallèle avec celui qui l'a appelé (thread principal ou secondaire), mais dans le **même** processus !

Le dernier argument permet de passer un paramètre à la fonction, ou plusieurs regroupés dans une structure.

Abandon, identification, égalité

Prototype abandon/fin :

```
void pthread_exit(void * retour);
```

Le paramètre `retour` est un résultat, *(valeur de retour)*, pouvant être consulté par un autre thread du même processus, attendant la fin de celui qui vient d'abandonner.

Prototype identification :

```
pthread_t pthread_self(void);
```

renvoie l'identité du thread appelant.

Prototype égalité :

```
int pthread_equal(pthread_t idT1, pthread_t idT2);
```

renvoie une valeur non nulle si réussite, 0 si échec.

Exemple

```
using namespace std;
#include <pthread.h>
#include <iostream>
#include <sys/types.h>

void *monThread (void * par){
  pthread_t moi = pthread_self();
  cout<< "thread " << moi << ", proc. "<< getpid()<< endl;
  pthread_exit(NULL);
}

int main(){
  pthread_t idTh;
  if (pthread_create(&idTh, NULL, monThread, NULL) != 0)
     cout << "erreur creation" <<endl;
     //suite...en particulier attendre la fin du thread!
}
```

Formes de synchronisation

Trois formes :

- Attendre la fin de l'exécution d'un thread,
- Exclusion mutuelle, via la notion de *verrous* binaires (à deux états, verrouillé ou non).
- Attendre l'occurrence d'un événement (pour imposer un ordre par exemple), via la notion de *variables conditionnelles*.

Synchronisation : problème 1

```
...
void *monThread (void * par){
  pthread_t moi = pthread_self();
  cout<< "monThread " << moi<< ", proc. "<<getpid()<< endl;
  calculDureeSec(3); // calcul qui dure 3 sec.
  cout<< "monThread : fin"<< endl;
  pthread_exit(NULL);
}
int main(){
  pthread_t idTh;
  if (pthread_create(&idTh, NULL, monThread, NULL) != 0)
     cout << "erreur creation" <<endl;
  cout<< "principal : fin"<< endl;
}
```

Question : que se passe-t-il à l'exécution ?

Attendre la fin de l'exécution d'un thread

Un thread quelconque, principal ou secondaire, peut attendre la fin d'un autre qu'il connaît :

Prototype

```
int pthread_join(pthread_t idT, void **retourCible);
```

Elle permet au thread appelant d'attendre la fin de celui issu du même processus, identifié par `idT`. Résultat 0 si réussite, $\neq$ 0 sinon.

L'appelant est bloqué si le thread *idT* n'est pas terminé. Il sera débloqué lorsque le thread *idT* aura fait `pthread_exit()`.

Retour à l'exemple

```
...
void *monThread (void * par){
  pthread_t moi = pthread_self();
  cout<< "monThread " << moi<< ", proc. "<<getpid()<< endl;
  calculDureeSec(3); // calcul qui dure 3 sec.
  cout<< "monThread : fin"<< endl;
  pthread_exit(NULL);
}
int main(){
  pthread_t idTh;
  if (pthread_create(&idTh, NULL, monThread, NULL) != 0)
     cout << "erreur creation" <<endl;
  int res = pthread_join(idTh, NULL);
  cout<< "principal : fin"<< endl;
}
```

Synchronisation : problème 2 (partie 1)

```
...
void *T1 (void * par){
  int * cp = (int*)(par);
  for(int i=0; i < 1500; i++)  ++(*cp);
  pthread_exit(NULL);
}

void *T2 (void * par){
  int * cp = (int*)(par);
  for(int i=0; i < 3000; i++) ++(*cp);
  pthread_exit(NULL);
}
```

Synchronisation : problème 2 (partie 2)

```
int main(){
  pthread_t idT1, idT2;
  int counter = 0;

  if (pthread_create(&idT1, NULL, T1, &counter) != 0)
     cout << "erreur creation" <<endl;
  if (pthread_create(&idT2, NULL, T2, &counter) != 0)
    cout << "erreur creation" <<endl;

  int res = pthread_join(idT1, NULL);
  res = pthread_join(idT2, NULL);
  std::cout<<" Total = "<<counter<<std::endl;

  return 0;
}
```

Question : Quel est le résultat final ?

Exclusion mutuelle

- Un *verrou* est un sémaphore ayant deux états possibles : **libre** ou **occupé** (verrouillé).
- Un thread peut demander de le verrouiller. Il peut obtenir ce verrouillage que si le verrou est libre et un seul thread peut obtenir le verrouillage.
- Lorsqu'un thread a obtenu le verrouillage, un autre thread qui demande le verrouillage du même verrou sera bloqué.
- Seul le thread qui a obtenu le verrouillage peut déverrouiller un verrou.
- Le verrouillage et déverrouillage sont des opérations atomiques.
- Un verrou est appelé `mutex`. Il est du type `pthread_mutex_t`.

Utilisation d'un verrou

Un mutex est une variable partagée entre threads. Il peut être une variable globale (non recommandé) ou une variable dont l'adresse sera passée en paramètre de chaque thread l'utilisant (solution à retenir).

Une fonction manipulant un mutex est de la forme
`pthread_mutex_fonction`

Fonction	Résultat
`pthread_mutex_init(...)`	verrou créé ; état libre
`pthread_mutex_lock(...)`	verrouillage
`pthread_mutex_trylock(...)`	verrouillage si état libre sinon, erreur sans blocage
`pthread_mutex_unlock(...)`	déverrouillage ; état libre
`pthread_mutex_destroy(...)`	destruction

Une autre façon d'initialiser un mutex :

```
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;
```

Retour à l'exemple

- Donner une solution au "problème 2"

Je retiens ...

- Tout accès à une variable accessible en lecture par un thread et en écriture par un autre doit être protégé.
- Un seul mutex peut protéger plusieurs variables, mais pas l'inverse.
- Les opérations *...lock()* et *...unlock()* sont atomiques, mais pas la portion de code qui se trouve entre les deux.
- Cette portion de code est appelée **section critique**
- Si un thread est dans une section critique, il doit être garanti qu'aucun autre thread n'y soit simultanément

Synchronisation : problème 3

Deux threads T_1, T_2 travaillent sur un entier x commun, T_1 effectue des opérations uniquement si $x > \mathit{seuil}$, et seule T_2 peut engendrer une telle situation.

Ce qu'on veut exprimer

T_1	T_2
...	...
tant que $X \leq$ *seuil* **faire**	modifier X
attendre;	**si** $X >$ *seuil* **alors**
	le signaler à T_1;
...	...

- y a t-il un problème d'exclusion mutuel à résoudre ?

Objectif

Thread 1

bloquer l'accès ;
tant que *prédicat non satisfait* **faire**
  relâcher accès;
  attendre event (prédicat satisfait);
  bloquer l'accès;
;
section critique
relâcher l'accès ;

Thread 2

bloque l'accès ;

section critique ;

signaler(prédicat satisfait) ;
relâcher l'accès ;

Avec :

- libération du verrou et passage à l'état bloqué de façon atomique,
- réveil en retrouvant le verrou.
- Remarque : plusieurs threads peuvent attendre la satisfaction d'un même prédicat et/ou le rendre insatisfait

Schéma de fonctionnement

- Un thread ayant réussi le verrouillage d'un verrou, pourra le relâcher et passer à l'état bloqué pour attendre un évènement, ceci de façon **atomique**. Il demandera aussi à être réveillé en retrouvant l'état verrouillé.
- Le réveil sera réalisé par un autre thread. Ce dernier peut activer un ou plusieurs threads en attente par l'intermédiaire d'une variable conditionnelle.
- Si plusieurs threads sont réveillés, un seul obtiendra le verrou, les autres ne le retrouveront que lorsqu'il sera déverrouillé.

Variable conditionnelle

Une *variable conditionnelle* est une donnée commune à plusieurs threads symbolisant l'occurrence d'un événement. Cet évènement annonce la satisfaction d'un ou de plusieurs prédicats.

Principe d'utilisation d'une variable conditionnelle

Partie Commune

```
int dCommune;
pthread_mutex_t verrou;
pthread_cond_t condi;
...// initialisations
```

Thread 1	Thread 2
.... ;	.... ;
pthread_mutex_lock(&verrou) ;	pthread_mutex_lock(&verrou) ;
tant que *(dCommune hors bornes)* **faire**	accèsEtModif(dCommune) ;
attendre(&condi, &verrou) ;	**si** *(dCommune dans bornes)* **alors**
;	réveillerTâches(&condi);
... // ici, je suis réveillé : travailler ;	;
pthread_mutex_unlock(&verrou) ;	pthread_mutex_unlock(&verrou);

Remarque : le prédicat ici n'est qu'une illustration !

Actions possibles

Sur une variable conditionnelle `c` et un verrou `v`, on peut effectuer les actions suivantes :

Fonction	Action
`pthread_cond_init(&c)`	crée la variable conditionnelle `c`
`pthread_cond_wait(&c,&v)`	bloque l'appelant **et** rend le verrou de façon atomique
`pthread_cond_timedwait (&c,&v,&délai)`	..._wait() avec attente temporelle
`pthread_cond_broadcast(&c)`	libère **tous** les threads bloqués
`pthread_cond_signal(&c)`	libère un seul thread
`pthread_cond_destroy(&c)`	destruction

Retour : Toutes ces fonctions retournent 0 (zéro) en cas de succès et un résultat non-nul en cas d'erreur, accompagné d'un code d'erreur.

Création, destruction

Prototype Création

```
int pthread_cond_init (          // résultat 0 si réussite, ≠ 0 sinon
pthread_cond_t *cond ,           // variable conditionnelle à créer
pthread_condattr_t *attr) ;      // NULL par défaut
```

Permet de créer une variable conditionnelle et de l'initialiser.
Note : pour des raisons de portabilité, utiliser l'initialisation par défaut.

Une initialisation plus simple peut être effectuée par la déclaration :

```
pthread_cond_t uneCond = PHTREAD_COND_INITIALIZER;
```

Prototype destruction

```
int pthread_cond_destroy(pthread_cond_ *cond);
```

Détruit la variable conditionnelle pointée.

Attente non bornée

Prototype :

```
int pthread_cond_wait (          // résultat 0 si réussite, ≠ 0 sinon
pthread_cond_t *vcond,           // variable conditionnelle associée
                                 // à l'évennement attendu
pthread_mutex_t *verrou) ;       // verrou d'accès à la donnée commune
```

Cette primitive réalise un appel bloquant, qui **de façon atomique** déverrouille `verrou` **et** attend que la condition `vcond` soit annoncée, forcément par un autre thread (voir ci-après pour l'annonce).

Note : La primitive suppose que le thread appelant a obtenu précédemment le verrouillage de `verrou`.

Attention : plusieurs threads peuvent être débloqués. Il est donc utile de tester à nouveau le prédicat après réveil (à la sortie de l'attente !)

Exemple

On reprend l'exemple de la tâche T_1 attendant que le prédicat $x > seuil$ soit vrai pour continuer son travail.

Partie Commune

```
int x, seuil;
pthread_mutex_t verrou;
pthread_cond_t condi;
...//initialisations diverses
```

Thread T_1

.... ;
pthread_mutex_lock(&verrou) ;
tant que *(x ≤ seuil)* **faire**
  pthread_cond_wait(&condi, &verrou) ;
;
... // ici, je suis réveillé et le verrou verrouillé : travail ;
pthread_mutex_unlock(&verrou) ;

Remarques

- L'attente provoque le déverrouillage de `verrou`, donc un autre thread peut le verrouiller,
- Tous les threads attendant sur la même variable conditionnelle doivent spécifier le même verrou dans l'attente : une variable conditionnelle est associée à un et un seul verrou (mutex), mais un verrou peut être associé à un nombre quelconque de conditions,
- la variable conditionnelle `condi` est utilisée comme un avertisseur (un drapeau) : lorsqu'on est averti, il s'est passé un événement sur lequel on a demandé un réveil.

Annonces

Il y a deux façons permettant de réveiller des threads en attente sur une condition :

- réveiller tous les threads en attente,
- réveiller un seul thread, mais ce sera un parmi ceux qui attendent, sans pouvoir choisir.

Prototype 1

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

provoque le réveil d'un seul thread.

Attention : il n'y a aucun rapport entre `signal` dans la fonction de réveil de thread vu ici et le *signal* vu en cours de système comme une forme d'interruption logicielle.

Annonces - suite

Prototype 2

`int pthread_cond_broadcast(pthread_cond_t *cond);`
provoque le réveil de tous les threads attendant la variable conditionnelle `cond`.

Important : Lors du réveil, les threads réveillés vont obtenir tour à tour automatiquement le verrouillage du verrou.

Exemple

On reprend l'exemple du thread T_1 attendant que le prédicat $x > seuil$ rende vrai pour continuer son travail. Ici, le cas de T_2 qui réveille T_1.

Tâche T_2

```
.... ;
pthread_mutex_lock(&verrou) ;
.... ; //travail sur x et/ou seuil ;
si (x > seuil) alors
  pthread_cond_broadcast(&condi) ;
;
pthread_mutex_unlock(&verrou) ;
```

Question : que se passe-t-il si aucun thread n'attend ?
Réponse : le réveil est **perdu** ! C'est logique, mais suppose que tous les threads testent le prédicat **avant** d'attendre.

`signal` **ou** `broadcast` ?

Recommandations :

- Utiliser `signal` seulement si on est certain que **n'importe** quelle tâche en attente peut faire le travail requis **et** qu'il est indispensable de réveiller une seule tâche.
- Lorsqu'une variable conditionnelle est utilisée pour plusieurs prédicats, `signal` est à prohiber.
- En cas de doute, utiliser `broadcast`.

Déverrouiller après ou avant l'annonce ?

- **Après** engendre qu'un thread réveillé ne pourra pas obtenir le verrouillage immédiatement car le verrou est toujours indisponible.
- **Avant** peut être plus efficace, mais il se peut aussi qu'un thread T_z non (encore) en attente obtienne le verrouillage. Il n'y a pas d'équité, alors que le thread réveillé T_a peut être plus prioritaire (T_z moins prioritaire a obtenu le verrouillage alors que T_a, en attente de l'annonce, ne pouvait l'obtenir).
- En conclusion, le choix dépendra du contexte.

Exemple : producteur - consommateur

- Un producteur dépose des messages qui seront extraits par un consommateur.
- Les communications se font à travers une file (FIFO) circulaire commune de taille limitée.
- Le producteur ne peut déposer un message dans la file tant que la file est pleine.
- Le consommateur ne peut extraire un message de la file tant que la file est vide.

Exemple : déclarations et initialisations

```
#include <stdio.h>
#include <pthread.h>
#define BUFFER_SIZE 16

/* circular buffer of integers */
struct prodcons {
  int buffer [BUFFER_SIZE ]; /* the actual data */
  int read_pos, write_pos;   /* positions for read and write */
  pthread_mutex_t lock;      /* mutex ensuring exclusive */
                             /* access to buffer */
  pthread_cond_t notempty;   /* signaled when buffer is not empty */
  pthread_cond_t notfull;    /* signaled when buffer is not full */
} buffer;

/* Initialize a buffer */
void init(struct prodcons * b){
  pthread_mutex_init(&(b->lock), NULL);
  pthread_cond_init(&(b->notempty), NULL);
  pthread_cond_init(&(b->notfull), NULL);
  b->read_pos = 0;
  b->write_pos = 0;
}
```

Exemple : retrait d'un message

```
int get(struct prodcons *b){
  int data;
  pthread_mutex_lock(&b->lock);
  while (b->write_pos == b->read_pos) {
    /* Wait until buffer is not empty */
    pthread_cond_wait(&b->notempty, &b->lock);
  }
  data = b->buffer[b->read_pos];
  b->read_pos++;
  if (b->read_pos >= BUFFER_SIZE)
    b->read_pos = 0;

  // signal that the buffer is now not full
  pthread_cond_signal(&b->notfull);
  pthread_mutex_unlock(&b->lock);
  return data;
}
```

Exemple : dépôt d'un message

```
void put(struct prodcons *b, int data){
  pthread_mutex_lock(&b->lock);
  while ((b->write_pos + 1) % BUFFER_SIZE == b->read_pos){
    /* Wait until buffer is not empty */
    pthread_cond_wait(&b->notfull, &b->lock);
  }
  b->buffer[b->write_pos] = data;
  b->write_pos++;
  if (b->write_pos >= BUFFER_SIZE)
    b->write_pos = 0;

  // signal that the buffer is now not empty
  pthread_cond_signal(&b->notempty);
  pthread_mutex_unlock(&b->lock);
}
```

Exemple : producteur

```
#define OVER (-1)
void * producer (void * par){
  int n;
  for (int n = 0; n < 1000; n++){
    printf ("prod --> %d\n", n);
    put(&buffer, n);
  }

  put(&buffer, OVER);
  pthread_exit(NULL);
}
```

Exemple : consommateur

```
void * consumer (void * p){
  int d;
  while (1){
    d = get (&buffer);
    if (d == OVER) break;
    printf ("cons --> %d\n", d);
  }
  pthread_exit(NULL);
}
```

Exemple : programme principal

```
int main (){
  pthread_t th_p, th_c;
  void * retval;
  init (&buffer);
  /* Create the threads */
  pthread_create (&th_p, NULL, producer, NULL);
  pthread_create (&th_c, NULL, consumer, NULL);
  /* Wait until producer and consumer finish */
  pthread_join (th_p, &retval);
  pthread_join (th_c, &retval);
  // remarque : destruction à faire ici.
  return 0;
}
```