Les processus légers : *threads*

Louis Mandel
louis.mandel@lri.fr
Laboratoire de Recherche en Informatique
Université Paris-Sud 11

IFIPS
Cycle ingénieur de la filière étudiant
année 2007/2008

Les threads

- Les threads sont des processus légers exécutés « à l'intérieur » d'un processus
- L'exécution des threads est concurrente
- Il existe toujours au moins un thread : le thread principal
- La durée de vie d'un thread ne peut pas dépasser celle du processus qui l'a crée
- Les threads d'un même processus partagent la même mémoire

Threads préemptifs vs Threads coopératifs

- Threads préemptifs
  - le système peut suspendre l'exécution d'un thread à tout moment pour exécuter un autre thread
  - Problème du non-déterminisme
- Threads coopératifs
  - chaque thread décide quand il peut être suspendu
  - Problème de réactivité (un thread ne rend jamais la main !)

- cette semaine : threads préemptifs
- dernier cours : threads coopératifs

Les threads Posix : Pthread

- La création
  - `#include <pthread.h>`

    ```
    int pthread_create(pthread_t *pthread,
                       const pthread_attr_t *attr,
                       void *(*fonction)(void *), void *arg);
    ```
  - `thread` : pour récupérer l'identité du thread qui est créé
  - `attr` : attributs du thread. Si `attr` est `NULL`, la valeur par défaut est prise
  - `fonction` : fonction à exécuter en parallèle
  - `arg` : arguments de la fonction
- Identité d'un thread
  - `pthread_t pthread_self(void);`
- Égalité ente threads
  - `int pthread_equal(pthread_t t1, pthread_t t2);`

Exemple : gcc thread-1.c -lpthread (thread-1.c)

```
#include <stdio.h>
#include <pthread.h>
pthread_t tid[3];
char *nom[3] = { "ainee", "cadette", "benjamine" };
void *annexe(void *arg) {
  int boucle;
  printf("Je suis la Pthread %s d'identite %d\n",
 (char *)arg, (int)pthread_self());
  for(boucle=0; boucle < 10000000; boucle++);
  printf("%s: je suis apres le premiere boucle\n", (char *)arg);
  for(boucle=0; boucle < 10000000; boucle++);
  printf("%s: je suis apres le deuxieme boucle\n", (char *)arg);
  return NULL; }
int main() {
  int ind, rep;
  printf("Processus %d lance\n", getpid());
  for (ind=0; ind<3; ind++) {
    if ( (rep = pthread_create(tid+ind, NULL, annexe, nom[ind])) == 0) {
      printf("Pthread %d creee\n", ind);
    } else { fprintf(stderr, "%d : ", ind); perror("pthread_create"); }
  }
  sleep(20); }
```

Exemple (thread-2.c)

- durée de vie limitée à celle du processus

```
#include <stdio.h>
#include <pthread.h>
void *immortelle(void *arg) {
  while (1) { printf("coucou\n"); }
}
int main() {
  pthread_t tid;
  int rep;
  if ( (rep = pthread_create(&tid, NULL, immortelle, "")) == 0) {
      printf("Pthread creee\n");
    } else { perror("pthread_create"); }
  sleep(1);
}
```

Terminaison (thread-4.c)

```
void *f(void *arg) {
  exit(1);
}
int main() {
  pthread_t tid;
  int rep;
  if ( (rep = pthread_create(&tid, NULL, f, "")) == 0) {
    printf("Pthread creee\n");
  } else { perror("pthread_create"); }
  sleep(5);
  printf("FIN"); }
```

- Le message FIN est-il affiché ?

Terminaison

- Terminaison d'un thread
  - `void pthread_exit(void *value_ptr);`
    - Si le thread a l'attribut `PTHREAD_CREATE_JOINABLE`, à sa mort, le thread devient un « zombie » jusqu'à ce qu'un autre thread en prenne connaissance
    - Si le thread a l'attribut `PTHREAD_CREATE_DETACHED`, à sa mort, le thread disparaît directement
- L'attribut `detachstate`
  - il peut être fixé à la création du thread
  - être modifié : `int pthread_detach(pthread_t thread);`

Terminaison (thread-5.c)

```
void *f(void *arg) {
  pthread_exit(NULL);
}
int main() {
  pthread_t tid;
  int rep;
  if ( (rep = pthread_create(&tid, NULL, f, "")) == 0) {
    printf("Pthread creee\n");
  } else { perror("pthread_create"); }
  sleep(5);
  printf("FIN"); }
```

Synchronisation

- Attendre la fin d'un thread
  - `int pthread_join(pthread_t tid, void **value)`
  - Appel bloquant jusqu'à ce que le thread `tid` termine (ou soit résilié, cf `pthread_cancel`)
  - Si `value` n'est pas `NULL` pointeur vers la valeur de retour du thread (ou, pointeur vers `PTHREAD_CANCELED` en cas de résiliation)
  - `tid` ne doit pas être détaché
  - `pthread_join` revoie 0 en cas de succès (`errno` n'est pas positionné en cas d'échec)

Exemple (thread-6.c)

```
pthread_t tid[3];
void *annexe(void *arg) {
  int boucle;
  printf("Je suis la Pthread %d d'identite %d\n",
 *((int *)arg), (int)pthread_self());
  *((int *)arg) =  (*((int *)arg) + 1);
  pthread_exit(arg); }
int main() {
  int ind, rep;
  int *valeur;
  printf("Processus %d lance\n", getpid());
  for (ind=0; ind<3; ind++) {
    int *pi = malloc(sizeof(int)); *pi=ind;
    if ( (rep = pthread_create(tid+ind, NULL, annexe, pi)) == 0) {
      printf("Pthread %d creee\n", ind);
    } else { fprintf(stderr, "%d : ", ind); perror("pthread_create"); }
  }
  for (ind=0; ind<3; ind++) {
    pthread_join(tid[ind], (void **) &valeur);
    printf("Fin de la Pthread d'identite %d et de valeur %d\n",
   (int)tid[ind], *valeur);
  } }
```

Mémoire partagée (conc-1.c)

```
int cpt = 0;
void *incr(void *arg) {
  int i;
  printf("[%d]: adr de i = %p, adr de cpt = %p\n", (int)pthread_self(), &i, &cpt);
  for(i=0; i < 1000000; i++) {
    if ( i % 10000 == 0) { printf("[%d]: i = %d\n", (int)pthread_self(), i); }
    cpt++;
  }
  printf("[%d]: FIN\n", (int)pthread_self());
  return NULL;
}
int main() {
  pthread_t tid[2];
  pthread_create(tid, NULL, incr, NULL);
  pthread_create(tid+1, NULL, incr, NULL);
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("cpt = %d", cpt);
}
```

Sections critiques

- Les accès en lecture et en écriture à des données partagées doivent être limités à un seul thread à la fois :
  - les threads doivent être en exclusion mutuelle
- Les parties du code qui doivent être accédées par au plus un processus sont des sections critiques
- Pour que des threads coopèrent correctement et efficacement, ils doivent respecter les conditions suivantes :
  1. Deux processus ne peuvent pas être simultanément dans une section critique relative à une ressource commune
  2. Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs
  3. Aucun processus suspendu en dehors en dehors de sa section critique ne doit bloquer les autres
  4. Aucun processus ne doit attendre trop longtemps avant de pouvoir entrer dans sa section critique

Exclusion mutuelle par accès atomiques en mémoire

- La programmation de l'exclusion mutuelle est difficile (au moyen de la seule indivisibilité des accès en mémoire)
- Plusieurs solutions erronées ont été publiées
  - Pour montrer qu'une solution est fausse, on construit un scénario avec des entrelacements d'opérations qui conduisent à un problème
- Les preuves de corrections sont difficiles
  - Première solution prouvée correcte publiée par Deckker et Dijkstra en 1966
  - Peterson à publié une solution plus simple en 1981
- Solutions reposant sur de l'attente active
  - test répétitifs des variables utilisées pour la synchronisation

Solution 1 (fausse)

- Variables partagées : `isc1`, `isc2` (vraies si intention d'entrer en section critique)
- Initialement : `isc1 = FAUX`, `isc1 = FAUX`

Processus 1	Processus 2

`while (VRAI) {`
`  isc1 = VRAI;`
`  while (isc2) { ; }`
`  // section critique ...`
`  isc1 = FAUX;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  isc2 = VRAI;`
	`  while (isc1) { ; }`
	`  // section critique ...`
	`  isc2 = FAUX;`
	`  // section restante`
	`done`

- Scénario fautif ?

Solution 1 (fausse)

- Variables partagées : `isc1`, `isc2` (vraies si intention d'entrer en section critique)
- Initialement : `isc1 = FAUX`, `isc1 = FAUX`

Processus 1	Processus 2

`while (VRAI) {`
`  isc1 = VRAI;`
`  while (isc2) { ; }`
`  // section critique ...`
`  isc1 = FAUX;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  isc2 = VRAI;`
	`  while (isc1) { ; }`
	`  // section critique ...`
	`  isc2 = FAUX;`
	`  // section restante`
	`done`

- Scénario fautif : P1 affecte `isc1 = VRAI` ; P2 affecte `isc1 = VRAI` ;
- $\Rightarrow$ interblocage

Solution 2 (fausse)

- Variables partagées : `sc1`, `sc2` (vraies si en section critique)
- Initialement : `sc1 = FAUX`, `sc1 = FAUX`

Processus 1	Processus 2

`while (VRAI) {`
`  while (sc2) { ; }`
`  sc1 = VRAI;`
`  // section critique ...`
`  sc1 = FAUX;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  while (sc1) { ; }`
	`  sc2 = VRAI;`
	`  // section critique ...`
	`  sc2 = FAUX;`
	`  // section restante`
	`done`

- Scénario fautif ?

Solution 2 (fausse)

- Variables partagées : `sc1`, `sc2` (vraies si en section critique)
- Initialement : `sc1 = FAUX`, `sc1 = FAUX`

Processus 1	Processus 2

`while (VRAI) {`
`  while (sc2) { ; }`
`  sc1 = VRAI;`
`  // section critique ...`
`  sc1 = FAUX;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  while (sc1) { ; }`
	`  sc2 = VRAI;`
	`  // section critique ...`
	`  sc2 = FAUX;`
	`  // section restante`
	`done`

- Scénario fautif :
  - P1 test `sc2` à faux ; P2 test `sc1` à faux
  - P1 affecte `sc1 = VRAI` ; P2 affecte `sc1 = VRAI` ;
  - P1 et P2 sont en même temps en section critique !

Solution 3 (fausse)

- Variables partagées : `tour`
- Initialement : `tour = 1`

Processus 1	Processus 2

`while (VRAI) {`
`  while (tour != 1) { ; }`
`  // section critique ...`
`  tour = 2;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  while (tour != 2) { ; }`
	`  // section critique ...`
	`  tour = 1;`
	`  // section restante`
	`done`

- Scénario fautif ?

Solution 3 (fausse)

- Variables partagées : `tour`
- Initialement : `tour = 1`

Processus 1	Processus 2

`while (VRAI) {`
`  while (tour != 1) { ; }`
`  // section critique ...`
`  tour = 2;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  while (tour != 2) { ; }`
	`  // section critique ...`
	`  tour = 1;`
	`  // section restante`
	`done`

- Scenario fautif : P1 stoppé hors section critique
- $\Rightarrow$ P2 ne peut pas entrer en section critique

Solution de Peterson

```
#define N 2
#define VRAI 1
#define FAUX 0
int tour; /* a qui le tour */
int interesse[N] = { FAUX, FAUX }; /* initialise a FAUX */

void entrer_region(int process) {
  int autre = 1-process;
  interesse[process] = VRAI;
  tour = autre;
  while ( (interesse[autre] == VRAI) && (tour == autre) ) { ; }
}
void quitter_region(int process) {
  interesse[process] = FAUX;
}
```

Attente passive : les mutex

- L'attente active n'est jamais la bonne solution
  - On préfère endormir un thread plutôt que faire de l'attente active
  - On devra alors être capable de le réveiller
- Les mutex
  - un mutex peut être libre ou verrouillé
  - un thread peut obtenir le verrouillage d'un mutex et en devenir propriétaire
  - un mutex ne peut être verrouillé que par un seul thread à la fois
  - toute demande de verrouillage d'un mutex déjà verrouillé entraîne le blocage du thread qui fait la demande ou l'échec de la demande

Les mutex

- Un mutex est une variable de type `pthread_mutex_t`
- Initialisation d'un mutex
  - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
- Verrouillage d'un mutex
  - `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- Déverrouillage d'un mutex
  - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Exemple (mutex.c)

```
int cpt = 0;
pthread_mutex_t mutex =PTHREAD_MUTEX_INITIALIZER;
void *incr(void *arg) {
  int i;
  printf("[%d]: adr de i = %p, adr de cpt = %p\n", (int)pthread_self(), &i, &cpt);
  for(i=0; i < 1000000; i++) {
    if ( i % 10000 == 0) { printf("[%d]: i = %d\n", (int)pthread_self(), i); }
    pthread_mutex_lock(&mutex);
    cpt++;
    pthread_mutex_unlock(&mutex);
  }
  printf("[%d]: FIN\n", (int)pthread_self());
  return NULL;
}
int main() {
  pthread_t tid[2];
  pthread_create(tid, NULL, incr, NULL);
  pthread_create(tid+1, NULL, incr, NULL);
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("cpt = %d", cpt); }
```

Le problème des philosophes (Dijkstra 1965)

- Cinq philosophes sont assis autour d'une table ronde
- Chaque philosophe a devant lui un plat de spaghettis tellement glissants qu'il faut deux fourchettes pour pouvoir les manger
- Une fourchette sépare chaque plat (il y a donc autant de plats que de philosophes que de fourchettes)
- Le comportement de chaque philosophe est le suivant :

```
while (1) {
  penser();
  obtenir_fourchettes();
  manger();
  relacher_fourchettes();
}
```

- Les fourchettes représentent des ressources que le processus doit détenir exclusivement pour pouvoir continuer à travailler

Le problème des philosophes

- Écrire le code des fonctions `obtenir_fourchettes` et `relacher_fourchettes`, de telle sorte que les contraintes suivantes soit satisfaites :
  - Un seul philosophe à la fois peut détenir une fourchette
  - Les processus ne doivent pas arriver à un interblocage
  - Plusieurs philosophes doivent pouvoir manger en même temps
  - Un philosophe ne doit mourir de faim en attendant une fourchette
- On donne le code suivant :

```
#define NbPhilo 5
int gauche(int i) { return (i+1)%NbPhilo; }
int droite(int i) { return (i+1); }
pthread_mutex_t fourchettes[NbPhilo];
```

- les mutex sont initialisés

```
for (i=0; i < NbPhilo; i++) { pthread_mutex_init(&fourchettes[i], NULL); }
```