Les processus légers : *threads*

Polytech Paris-Sud

Cycle ingénieur de la filière étudiant

Louis Mandel

Université Paris-Sud 11

Louis.Mandel@lri.fr

année 2011/2012

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 !)

- dans ce cours : threads préemptifs

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);
    ```
  - `pthread` : 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-0.c -lpthread

thread-0.c

```
void *f(void *arg) {
  while (1) { printf("Je suis la Pthread\n"); }
  return NULL;
}

int main() {
  pthread_t tid;
  int err;
  printf("Processus %d lance une Pthread\n", getpid());
  if ( 0 != (err = pthread_create(&tid, NULL, f, NULL)) ) {
    fprintf(stderr, "pthread_create: %s", strerror(err));
    exit(EXIT_FAILURE);
  }
  printf("Pthread creee\n");
  while (1) { printf("Je suis la fonction main\n"); }
  return 0;
}
```

Exemple : gcc thread-1.c -lpthread

thread-1.c

```
void *annexe(void *arg) {
  int boucle;
  printf("Je suis la Pthread %s d'identite %lu\n",
         (char *)arg, (unsigned long)pthread_self());
  for(boucle=0; boucle < 10000000; boucle++);
  printf("%s: je suis apres la premiere boucle\n", (char *)arg);
  for(boucle=0; boucle < 10000000; boucle++);
  printf("%s: je suis apres la deuxieme boucle\n", (char *)arg);
  return NULL;
}

int main() {
  pthread_t tid[3];
  char *nom[3] = { "ainee", "cadette", "benjamine" };
  int ind;
  printf("Processus %d lance\n", getpid());
  for (ind=0; ind<3; ind++) {
    if ( 0 == pthread_create(&(tid[ind]), NULL, annexe, nom[ind]) ) {
      printf("Pthread %d creee\n", ind);
    } else { fprintf(stderr, "pthread_create: %d", ind); }
  }
  sleep(5);
  return 0;
}
```

Exemple

thread-2.c

- durée de vie limitée à celle du processus

```
void *immortelle(void *arg) {
  while (1) { printf("coucou\n"); }
}


int main() {
  pthread_t tid;
  int err;
  if ( 0 != (err = pthread_create(&tid, NULL, immortelle, NULL)) ) {
    fprintf(stderr, "pthread_create: %s", strerror(err));
    exit(EXIT_FAILURE);
  }
  printf("Pthread creee\n");
  sleep(1);
  return 0;
}
```

Terminaison

thread-4.c

```
void *f(void *arg) {
  exit(1);
}

int main() {
  pthread_t tid;
  int err;
  if ( 0 != (err = pthread_create(&tid, NULL, f, NULL)) ) {
    fprintf(stderr, "pthread_create: %s", strerror(err));
    exit(EXIT_FAILURE);
  }
  sleep(5);
  printf("FIN\n");
  return 0;
}
```

- 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 err;
  if ( 0 != (err = pthread_create(&tid, NULL, f, NULL)) ) {
    fprintf(stderr, "pthread_create: %s", strerror(err));
    exit(EXIT_FAILURE);
  }
  sleep(5);
  printf("FIN\n");
  return 0;
}
```

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 l'argument `value` n'est pas `NULL`, il pointe vers la valeur de retour du thread (ou, pointe 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

```
void * succ(void * arg) {
  int i = *((int *) arg);
  int * res;
  if (NULL == (res = (int *) malloc(sizeof(int))) ) {
    perror("malloc"); exit(EXIT_FAILURE);
  }
  *res = i+1;
  pthread_exit(res);
}

int main() {
  pthread_t tid;
  int err;
  int arg = 0;
  int *res;
  if ( 0 != (err = pthread_create(&tid, NULL, succ, &arg)) ) {
    fprintf(stderr, "pthread_create: %s", strerror(err)); exit(EXIT_FAILURE);
  }
  pthread_join(tid, (void **) &res);
  printf("%d + 1 = %d\n", arg, *res);
  return 0;
}
```

Mémoire partagée

conc-1.c

```
int cpt = 0;
void *incr(void *arg) {
  int i;
  printf("[%lu]: adr de i = %p, adr de cpt = %p\n",
         (unsigned long)pthread_self(), &i, &cpt);
  for(i=0; i < 1000000; i++) { cpt++; }
  printf("[%lu]: FIN\n", (unsigned long)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\n", cpt);
  return 0;
}
```

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`, `isc2 = FAUX`

Processus 1	Processus 2

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

- Scénario fautif ?

Solution 1 (fausse)

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

Processus 1	Processus 2

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

- 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`, `sc2 = FAUX`

Processus 1	Processus 2

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

- Scénario fautif ?

Solution 2 (fausse)

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

Processus 1	Processus 2

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

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

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

- Scénario 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

- 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
  - ```
    int pthread_mutex_init(pthread_mutex_t *mutex,
                           pthread_mutexattr_t *attr);
    ```
- Verrouillage d'un mutex
  - ```
    int pthread_mutex_lock(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;

void *incr(void *arg) {
  int i;
  for(i=0; i < 1000000; i++) {
    pthread_mutex_lock(&mutex);
    cpt++;
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}

int main() {
  pthread_t tid[2];
  pthread_mutex_init (&mutex, NULL);
  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);
  return 0;
}
```

Les sémaphores

- Sémaphores
  - Un compteur
  - Une file d'activités en attente
- Compteur associé au sémaphore
  - nombre de jetons
  - valeur positive = nombre d'activités pouvant acquérir librement la ressource
  - valeur négative = nombre d'activités blocquées en attente de la ressource

Les sémaphores

- Primitives pour l'utilisation des sémaphores
  - Création / initialisation : nombre initial de jetons
  - Prendre
    - classiquement noté P (puis-je ?)
    - attribue un jeton à l'activité demandeuse s'il en reste un
    - la bloque sinon
  - Libérer
    - classiquement noté V (vas-y !)
    - rend un jeton
    - débloque une activité s'il en existe une en attente

Syntaxe des sémaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

Exclusion mutuelle avec les sémaphores

```
struct mutex {
  sem_t mutex_sem;
};

void mutex_init(struct mutex *m) {
  sem_init(m.mutex_sem, 0, 1);
}
void mutex_lock(struct mutex *m) {
  sem_wait(m.mutex_sem);
}
void mutex_unlock(struct mutex *m) {
  sem_post(m.mutex_sem);
}
```