Programmation Concurrente

Principes et concepts
2024-2025

Chapitre7: Variables de condition

Plan

- Définition Variable de condition
- Implémentation
- Exemples
- Limites

Condition

- Un thread peut vouloir exécuter un certain code seulement si une ou plusieurs conditions sont remplies
- Si on peut exprimer une condition dans un programme, alors on peut utiliser une variable de condition
- Une variable de condition est un mécanisme permettant de tester une condition pour laquelle:
    - le thread est bloqué (attente passive) tant que la condition n'est pas satisfaite;
    - un ou plusieurs thread(s) est/sont réveillé(s), si la condition devient vraie.

Variable de condition

- Une variable de condition est une file d'attente de thread(s) en attente sur une condition à l'intérieur d'une section critique.
- Idée principale: permet de bloquer (attente passive) à l'intérieur de la section critique en relâchant le verrou de manière atomique au moment de la mise en attente.
- Opérations:
  - wait(&lock): relâche le verrou et bloque le thread, le tout atomiquement. Lorsque la condition est ensuite signalée, le thread reprend son exécution, mais en reverrouillant le verrou avant de continuer.
  - signal: réveille un des threads en attente sur la condition.
  - broadcast: réveille tous les threads en attente sur la condition.

Variable de condition

- Une variable de condition est une file d'attente de thread(s) en attente sur une condition à l'intérieur d'une section critique.
- Idée principale: permet de bloquer (attente passive) à l'intérieur de la section critique en relâchant le verrou de manière atomique au

Pour toute opération sur une variable de condition, le verrou doit être préalablement verrouillé !

reprend son exécution, mais en re-verrouillant le verrou avant de continuer.

- signal: réveille un des threads en attente sur la condition.
- broadcast: réveille tous les threads en attente sur la condition.

Interface(1)

- Interface de gestion
  - Type de donnée: **pthread_cond_t**
  - Création:
    - Statique: **pthread_cond_t** cond = **PTHREAD_COND_INITIALIZER** ;
    - Dynamique: int **pthread_cond_init**(**pthread_cond_t** *, pthread_condattr t *)
  - Destruction: int **pthread_cond_destroy**(**pthread_cond_t** *)

Interface (2)

- Interface d'usage
  - **pthread_cond_signal**(**pthread_cond_t** *) Débloque un éventuel thread bloqué
  - **int pthread_cond_wait**(**pthread_cond_t** *cond, **pthread_mutex_t** *mutex) Bloque tant que la condition n'est pas réalisé
  - **int pthread_cond_timedwait**(**pthread_cond_t** *cond, **pthread_mutex_t** *mutex, **const struct** timespec *abstime) Bloque au plus tard jusqu'à la date spécifiée
  - **Pthread_cond_broadcast**(pthread cond t *) Réveille tous les bloqués
- Pièges des conditions
  - Risque d'interblocage ou pas de blocage si protocole du mutex n'est pas bien suivi
  - Signaler une condition que personne n'attend

Initialisation

```
// Initialisation statique
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// Initialisation dynamique
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

- Passer un attribut de condition `NULL` à `pthread_cond_init` permet d'utiliser les attributs par défaut.
- L'implémentation Linux ne supporte pas d'attribut et ignore ce 2ème argument.
- Toutes les fonctions sur les variables de condition renvoient 0 en cas de succès et une valeur différente de 0 en cas d'erreur.

Attente

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

- Effectue les opérations suivantes, de manière atomique:
  - **relâche** le verrou `mutex`;
  - attend que la variable de condition `cond` soit signalée.
- L'exécution du thread est suspendue (attente passive) jusqu'à ce que `cond` soit signalée.
- Le `mutex` doit être **verrouillé** par le thread **avant** l'appel à `pthread_cond_wait`.
- Au moment où `cond` est signalée, `pthread_cond_wait` **re-verrouille** automatiquement le `mutex`.

Signalisation

```
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

- **`pthread_cond_signal`** reveille un des threads en attente sur **`cond`**:
  - si aucun thread n'est en attente, la fonction n'a aucun effet
  - si plusieurs threads sont en attente, **un seul** est réveillé (choisi par l'ordonnanceur).
- **`pthread_cond_broadcast`** réveille **tous** les threads en attente sur **`cond`**:
  - si aucun thread n'est en attente, la fonction n'a aucun effet ;
  - les threads réveillés continuent leur exécution **chacun à leur tour**, car le mutex ne peut être repris que par un thread à la fois (l'ordre est **imprévisible** et dépend de l'ordonnanceur).

Exemple

```
void *child(void *arg) {
    // Comment indiquer que le
    // thread est terminé ?
    return NULL;
}

int main() {
    pthread_t t;
    pthread_create(&t,NULL,child,NULL);
    // Comment attendre la fin du
    // thread child ?
    return 0;
}
```

- Comment bloquer passivement le thread `main` tant que le thread `child` ne s'est pas terminé, sans utiliser de sémaphore, de barrière de synchronisation, ni d'appel à `pthread_join` ?
- Autrement dit, comment implémenter l'équivalent de `pthread_join` à l'aide de variables de condition ?

Exemple

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
       pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

Exemple

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
        pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

Deux cas possibles:

Deux possibilités

Cas-1

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

1

- Thread `main` créé le thread `child` et appelle `thr_join` avant que `child` ne s'exécute:
  - verrouille `m`;
  - teste `child_done` à faux;
  - bloque sur `cond_wait` (donc relâche `m`).
- Ensuite, le thread `child`:
  - verouille `m`;
  - met `child_done` à 1;
  - réveille (signal) `main`.
- Ensuite, le thread `main`:
  - reprend depuis l'appel à `cond_wait` avec `m` verrouillé;
  - puis relâche `m`;

Cas_2

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
      pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

2

- Le thread `child` s'exécute avant que `main` n'arrive pas à `thr_join`:
  - verouille `m`;
  - met `child_done` à 1;
  - signale `main`, ce qui n'a aucun effet (aucun thread bloqué).
- Ensuite, le thread `main`:
  - exécute `thr_join`;
  - verouille `m`;
  - teste `child_done` à vrai;
  - relâche `m`;
  - se termine.

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
      pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

- La variable `child_done` est-elle vraiment nécessaire ?

Test1

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
      pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

- La variable `child_done` est-elle vraiment nécessaire ?

Plan

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

- La variable `child_done` est-elle vraiment nécessaire ?
- Si le thread `child` est exécuté avant que `main` n'appelle `thr_join`:
  - la VC est signalée, mais cela n'a aucun effet, car aucun thread n'est bloqué
  - `main` va ensuite bloquer sur `cond_wait` dans la fonction `thr_join`...

  ⇨ **deadlock !**

test2

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

- L'utilisation du mutex `m` est-elle vraiment nécessaire ?

Test2

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(void *arg) {
   pthread_mutex_lock(&m);
   child_done = 1;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&m);
   return NULL;
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

- L'utilisation du mutex `m` est-elle vraiment nécessaire ?
- Si la fonction `thr_join` est exécutée avant le thread `child`:
  - `child_done` est testé à faux, mais le thread est préempté avant de faire le `wait`;
  - `child` prend la main, met `child_done` à 1 et signale la VC;
  - `thr_join` reprend la main et bloque dans l'appel à `wait`.

  ⇨ **deadlock !**

```
int child_done = 0;
pthread_mutex_t m =
          PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
          PTHREAD_COND_INITIALIZER;

void *child(
   pthread_m
   child_don
   pthread_c
   pthread_m
   return NU
}

void thr_join() {
   pthread_mutex_lock(&m);
   if (!child_done)
      pthread_cond_wait(&c,&m);
   pthread_mutex_unlock(&m);
}

int main() {
   pthread_t t;
   pthread_create(&t,NULL,child,NULL);
   thr_join();
   return 0;
}
```

Toujours verrouiller le mutex pendant

Les appels à **`pthread_cond_wait`**

et **`pthread_cond_signal`** **!**

- L'utilisation du mutex `m` est-elle vraiment nécessaire ?

`join` est

read `child`:

est testé à

thread est

nt de faire le

wait;

- `child` prend la main, met `child_done` à 1 et signale la VC;
- `thr_join` reprend la main et bloque dans l'appel à `wait`.

⇨ **deadlock !**

Plan

```
#define NUMb_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t     count_mutex; // Proteger le conteur
pthread_cond_t       count_threshold;

void *incr_count(void *t)
{
          int i;
          long mon_id = (long)t;
          for (i=0; i<TCOUNT; i++) {
                    pthread_mutex_lock(&count_mutex);
                    count++;

                    if (count == COUNT_LIMIT) {
                      pthread_cond_signal(&count_threshold);
                    }

                    pthread_mutex_unlock(&count_mutex);
                    usleep(1000); // forcer un peu le changement de contexte
           }
           pthread_exit(NULL);
}
void *recu_count(void *t)
{
          long my_id = (long)t;
          pthread_mutex_lock(&count_mutex);
          while (count<COUNT_LIMIT) {
                    pthread_cond_wait(&count_threshold, &count_mutex)
                    count += 125;
         }
         pthread_mutex_unlock(&count_mutex);
          pthread_exit(NULL);

}
```

```
int main (int argc, char *argv[])
{
          int i;
          long t1=1, t2=2, t3=3;
          pthread_t threads[3];

          pthread_mutex_init(&count_mutex, NULL);
          pthread_cond_init (&count_threshold, NULL);

          pthread_create(&threads[0], NULL, recu_count, (void *)t1);
          pthread_create(&threads[1], NULL, incr_count, (void *)t2);
          pthread_create(&threads[2], NULL, incr_count, (void *)t3);

          for (i=0; i<NUMb_THREADS; i++) {
            pthread_join(threads[i], NULL);
          }

          pthread_mutex_destroy(&count_mutex);
          pthread_cond_destroy(&count_threshold);
          pthread_exit(NULL);
}
```

Remarques finales

- Avantages des moniteurs
  - une protection associée au moniteur (exclusion mutuelle);
  - une souplesse d'utilisation des primitives attente et signale;
  - une efficacité de ces mécanismes.
- Inconvénients des moniteurs
  - un risque de manque de lisibilité qui est partiellement dû à des variations sémantiques des implémentations dans les divers langages qui les supportent.
    - Dans le cas de pthread, il n'y a aucune garantie que les variables partagées sont effectivement accédées uniquement depuis les points d'entrée du moniteur qui devrait les protéger;
  - les variables condition sont de bas niveau;
  - l'impossibilité d'imposer un ordre total ou partiel dans l'exécution des procédures ou fonctions exportées.