Programmation Concurrente

Principes et concepts
2024-2025

Chapitre5: Mutex et Barrières

Plan

- Rappel de l'attente active et ses inconvénients
- Définition des verrous(Mutex)
- Coordination de tâches à travers des Mutex
- Rendez-vous
- Limites
- Barrières de synchronisation

Spinlock et attente active

- Un spinlock, ou « verrou tournant » est un mécanisme d'exclusion mutuelle par attente active.
- Spinlocks implémentés grâce à des instructions matérielles spéciales comme « Test And Test » ou « Compare And Swap », etc
- La librairie pthread met à disposition:
  - le type pthread_spinlock_t
  - des fonctions de manipulation:
    - int pthread_spin_lock(pthread_spinlock_t *lock)
    - int pthread_spin_trylock(pthread_spinlock_t *lock)
    - int pthread_spin_unlock(pthread_spinlock_t *lock)

Attente active

- L'utilisation de variables pour l'exclusion mutuelle peut être ambiguë, rendant leur rôle confus.
- Une tâche en attente active monopolise inutilement le processeur, gaspillant des ressources en attendant une condition.

Un processeur peut être utilisé d'une façon plus efficace si il n'attend pas activement.

Mutex

- Un mutex (mutual exlusion) est un mécanisme d'exclusion mutuelle par attente passive permettant la gestion des threads, il a été introduit pour résoudre partiellement certains problèmes liés à la synchronisation des threads et à la gestion de données partagées.
- Un mutex est une structure de donnée constitué :
  - d'une variable booléenne
  - d'un propriétaire
  - d'une file d'attente
  - de deux opérations **atomiques** :
    - `lock` (vérouillage)
    - `unlock` (dévérouillage)

Type de mutex

- La librairie POSIX Threads défini 3 types de mutex:
  - mutex normal (ou rapide);
  - mutex sécurisé (*error checking);*
  - mutex récursif (*recursive*).

  https://docs.oracle.com/cd/E19455-01/806-5257/sync-78/index.html

- Le type d'un mutex défini le comportement à adopter en cas de verrouillage et déverrouillage.

Mutex normal (ou rapide)

- C'est le type de mutex par défaut dans POSIX.
- Un thread qui tente de le verrouiller alors qu'il est déjà verrouillé entre dans un état de blocage (il attend jusqu'à ce que le mutex soit libéré).
- Ce type de mutex ne détecte pas les erreurs d'utilisation (ex. un thread essayant de le déverrouiller sans l'avoir verrouillé auparavant).

- Utilisation typique : Protection simple de ressources partagées.

Mutex sécurisé (*error checking*)

- Ce type de mutex inclut des mécanismes de détection des erreurs.
- Il génère une erreur si un thread tente :
  - De déverrouiller un mutex qu'il n'a pas verrouillé.
  - De verrouiller un mutex qu'il a déjà verrouillé (ce qui éviterait un blocage involontaire).

- Utilisation typique : Débogage et validation de la gestion des mutex pour éviter des erreurs de synchronisation.

Mutex récursif (*recursive*)

- Permet à un même thread de verrouiller plusieurs fois le même mutex sans se bloquer.
- Chaque verrouillage doit être accompagné d'un nombre équivalent de déverrouillages pour libérer complètement le mutex.
- Utile pour éviter les deadlocks lorsque des fonctions imbriquées verrouillent le même mutex.

- Utilisation typique : Fonctions récursives nécessitant un verrouillage, ou structures logicielles complexes où un même thread peut acquérir plusieurs fois un mutex sans bloquer.

Pseudo-code

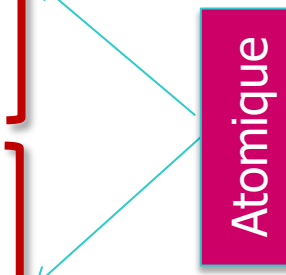
```
void lock(mutex m) {
    if (m.locked)
        bloque_thread_appelant_dans_file_associée_à_m();
    else
        m.locked = true;
}

void unlock(mutex m) {
    if (file_associée_à_m_est_vide())
        m.locked = false;
    else
        débloque_un_thread_dans_file_associée_à_m();
}
```

Pseudo-code

```
void lock(mutex m) {
    if (m.locked)
        bloque_thread_appelant_dans_file_associée_à_m();
    else
        m.locked = true;
}

void unlock(mutex m) {
    if (file_associée_à_m_est_vide())
        m.locked = false;
    else
        débloque_un_thread_dans_file_associée_à_m();
}
```



Définition des verrous (Mutex)

- Si un mutex est initialisé à *false* et qu'une tâche appelle lock(mutex), le mutex est positionné à *true*. Si une seconde tâche appelle mutex, alors cette tâche, qui est à l'état élu, passe à l'état bloqué et joint la file d'attente associée à mutex.

- La primitive unlock(mutex) réalise l'opération inverse. S'il y a une tâche en attente sur le mutex, alors cette tâche est réactivée. La tâche réveillée sort de la file d'attente associée pour joindre la file des tâches prêtes.

- L'exécution de la primitive unlock(mutex) revient à passer le mutex à la tâche réveillée. S'il n'y a pas de tâche en attente pour obtenir le mutex, la primitive unlock(mutex) libère le mutex en le positionnant à *false* (état initial).

Utilisation de mutex

```
mutex m;
mutex_init(m);
.
.
.
lock(m);
// section critique
unlock(m);
.
.
.
```

- Un verrou est un "gentlemen's agreement" qui permet de résoudre le problème de l'exclusion mutuelle à n tâches de manière simple.

Les mutex Pthreads

- Interfaces de gestion
  - Type de donnée: **Pthread_mutex_t**
  - Création:
    - statique: **Pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER ;**
    - dynamique: **Pthread_mutex_init**(pthread_mutex_t *, pthread_mutexattr_t *) ;

      Premier argument : adresse de (pointeur vers) la variable a initialiser
  - Destruction: **pthread_mutex_destroy**(mutex) (doit être déverrouillé)
- Interfaces d'usage
  - **Pthread_mutex_lock**(mutex) Bloque jusqu'à obtention du verrou
  - **Pthread_mutex_trylock**(mutex) Obtient le verrou ou renvoie EBUSY
  - **Pthread_mutex_unlock**(mutex) Libère le verrou

Exemple

```
pthread_mutex_t   mutex = PTHREAD_MUTEX_INITIALIZER;
int global=0;

void *tache(void *arg) {
    for(int i=0;i<1000000;i++) {
            pthread_mutex_lock(&mutex);
                global=global+1;
            pthread_mutex_unlock(&mutex);
    }
}
int main(void) {

    pthread_t tache1, tache2;

    if (pthread_create(&tache1,NULL,tache,(void *)1) == 0) {

        if (pthread_create(&tache2,NULL,tache,(void *)2) == 0) {

            pthread_join(tache1,NULL);
            pthread_join(tache2,NULL);
            printf("Fin des taches : global = %d (%d)\n",global, 2*iterations);

            return EXIT_SUCCESS;
         }
     printf("main: Erreur pthread_create\n");
}
printf("main: Erreur pthread_create\n");
return EXIT_FAILURE;
}
```

Synchronisation des threads

- Les verrous(Mutex) peuvent aussi servir à résoudre les problèmes liés à la coordination de tâches.

- Si I1 doit précéder I2, cette synchronisation se fait par le biais du verrou sync par contre si T2 est plus rapide que T1, T2 se bloque jusqu'à ce que T1 relâche le verrou sync

```
static pthread_mutex_t   sync =
                              PTHREAD_MUTEX_INITIALIZER;

void *T1(void *arg) {
        sleep(1);
        printf("T1: fini sleep\n");                    // I1
        pthread_mutex_unlock(&sync);
        return NULL;
} /* fin de T1 */

void *T2(void *arg) {
        printf("T2: avant pthread_mutex_lock\n");
        pthread_mutex_lock(&sync);
        printf("T2: apres pthread_mutex_lock\n");      // I2
        return NULL;
} /* fin de T2 */

int main(void) {
        pthread_t tache1, tache2;
        pthread_mutex_lock(&sync);
        if (pthread_create(&tache1,NULL,T1,NULL) == 0) {
                if (pthread_create(&tache2,NULL,T2, NULL) == 0) {
                        pthread_join(tache1,NULL);
                        pthread_join(tache2,NULL);
                        return EXIT_SUCCESS;
                }
        }
        return EXIT_FAILURE;
} /* fin de main
```

Rendez-vous(1)

- Rendez-vous : des processus collaborant doivent s'attendre mutuellement
  - Romeo et Juliette ne peuvent se prendre la main que s'ils se rencontrent
  - Processus devant échanger des informations entre les étapes de l'algorithme
- Chaque tâche dispose d'une activité qu'elle peut exécuter, suivie d'une autre activité pouvant être démarrée quand l'autre tâche a terminé sa première activité.

	TâcheA	TâcheB
Premières activités	a1	b1
Rendez-vous	X	
Deuxièmes activités	a2	b2

- Les activités a2 et b2 ne peuvent débuter que lorsque les activités a1 et b1 ont terminé. Ce type de synchronisation est un rendez-vous.
- Les 2 tâches ont un point de rencontre où aucune ne peut continuer avant que l'autre n'arrive à ce point.

Rendez-vous(2)

- L'exemple à droite demande un nombre élevé de changements de contexte. En effet, sur un monoprocesseur ou pour une exécution asynchrone, l'une des tâches arrivera Au point de rendez-vous avant l'autre.

1. Si TacheA arrive la première, la tâche se bloquera sur le verrou arriveB,
2. Puis lorsque TacheB termine b1, TacheB déverrouille arriveB avant de se bloquer sur arriveA.
3. La tâche TacheA peut alors reprendre son exécution et déverrouiller arriveA avant de progresser à son activité a2.

- Dans ce cas, il y a 3 changements de contexte. Si, au contraire TacheB termine b1 et arrive au point de rendez-vous avant TacheA, le fait de déverrouiller arriveB puis attendre sur arriveA, permet à TacheA de terminer son activité a1, puis de poursuive à a2 sans changement de contexte.

```
static pthread_mutex_t arriveA = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t arriveB = PTHREAD_MUTEX_INITIALIZER;
void *TacheA(void *arg) {
        a1;
        pthread_mutex_lock(&arriveB);
        pthread_mutex_unlock(&arriveA);
        a2;
        return NULL;
} /* fin de TacheA */
void *TacheB(void *arg) {
        b1;
        pthread_mutex_unlock(&arriveB);
        pthread_mutex_lock(&arriveA);
        b2;
        return NULL;
} /* fin de TacheB */
int main(void) {
        pthread_t tacheA, tacheB;
        pthread_mutex_lock(&arriveA);
        pthread_mutex_lock(&arriveB);
        if (pthread_create(&tacheA,NULL,TacheA,NULL) == 0) {
                if (pthread_create(&tacheB,NULL,TacheB,NULL) == 0) {
                        pthread_join(tacheA,NULL);
                        pthread_join(tacheB,NULL);
                        return EXIT_SUCCESS;
        }
        }
        return EXIT_FAILURE;
} /* fin de main */
```

Rendez-vous(3)

- L'exemple suivant minimise le nombre de changements de contexte. Ici, la dernière tâche qui arrive au point de rendez-vous informe l'autre via un déverrouillage, puis peut prendre le verrou positionné par l'autre tâche avant de poursuive.
- C'est uniquement la première tâche qui atteint le rendez-vous qui devra subir un changement de contexte.

```
static pthread_mutex_t arriveA = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t arriveB = PTHREAD_MUTEX_INITIALIZER;
void *TacheA(void *arg) {
        a1;
        pthread_mutex_unlock(&arriveA);
        pthread_mutex_lock(&arriveB);
        a2;
        return NULL;
} /* fin de TacheA */
void *TacheB(void *arg) {
        b1;
        pthread_mutex_unlock(&arriveB);
        pthread_mutex_lock(&arriveA);
        b2;
        return NULL;
} /* fin de TacheB */
int main(void) {
        pthread_t tacheA, tacheB;
        pthread_mutex_lock(&arriveA);
        pthread_mutex_lock(&arriveB);
        if (pthread_create(&tacheA,NULL,TacheA,NULL) == 0) {
                if (pthread_create(&tacheB,NULL,TacheB,NULL) == 0) {
                        pthread_join(tacheA,NULL);
                        pthread_join(tacheB,NULL);
                        return EXIT_SUCCESS;
        }
        }
        return EXIT_FAILURE;
} /* fin de
```

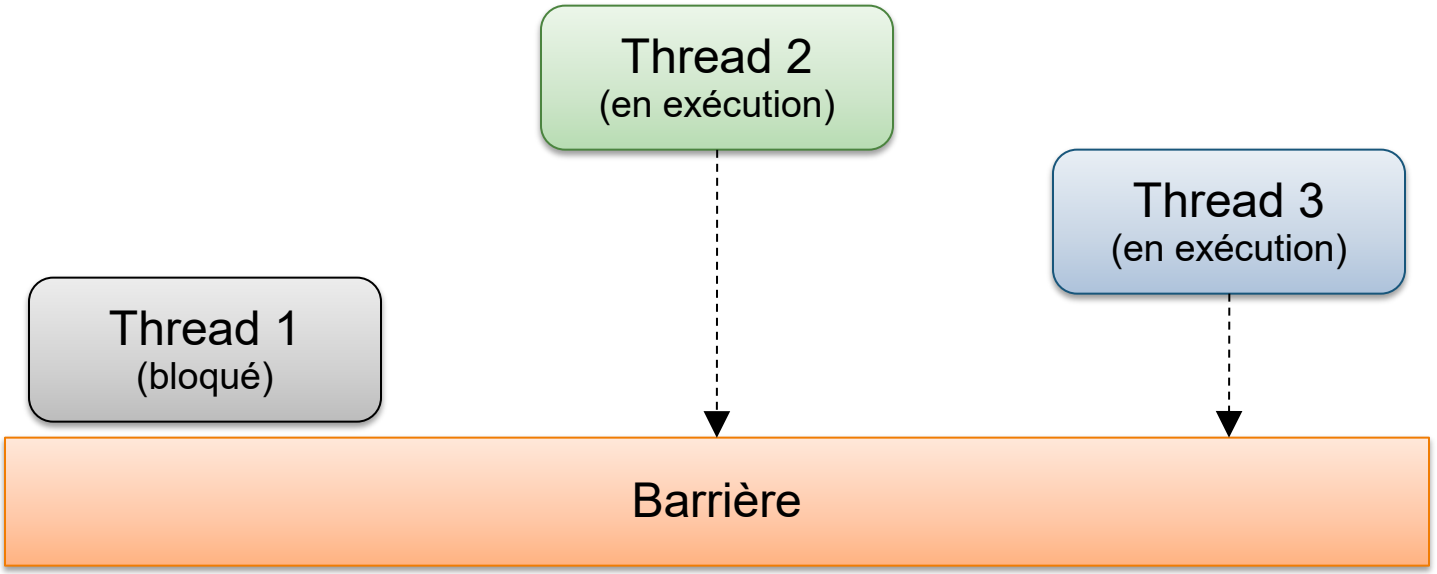
Verrouillage et performance

- En Une section critique implique une sérialisation ⇨ dégradation des performances !
- Toute section critique devrait être **la plus courte possible**
- Optimisation possible ?
- Optimisation:
  - `pthread_mutex_trylock` afin de ne pas bloquer et ainsi faire autre chose puis réessayer le verrouillage plus tard

Limites des verrous-Mutex

- En général si n tâche il faut n-1 boolean
- Le système peut devenir lourd avec plusieurs tâches →sinon recours au sémaphore.

Barrière de synchronisation



- Une barrière est un point de synchronisation pour *n* threads (ici 3)
- Aucun thread ne peut passer la barrière tant que tous n'y sont pas encore arrivés ⇨ attente passive

Fonctionnement d'une barrière

- Fonctionnement d'une barrière de synchronisation POSIX:
  1. Le nombre de threads à synchroniser est spécifié lors de la création de la barrière
  2. Chaque thread notifie son arrivée à la barrière
  3. Tant que tous les threads n'ont pas notifié la barrière, ceux déjà arrivés sont suspendus
  4. Une fois que les thread ont tous notifiés la barrière, celle-ci débloque tous les threads en attente (1 par 1 dans un ordre indéterminé)
  5. La barrière est ensuite réinitialisée à la valeur spécifiée lors de sa création

Utilisation des barrières

```
int pthread_barrier_init(pthread_barrier_t *barrier,
                         const pthread_barrierattr_t *attr, unsigned count);

int pthread_barrier_wait(pthread_barrier_t *barrier);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- `pthread_barrier_init()` créé une barrière:
  - Si `attr == NULL` ⇨ les attributs par défaut sont utilisés
  - le nombre de threads à se synchroniser est spécifié par `count`
- Chaque thread notifie sont arrivée avec `pthread_barrier_wait()`
- Les ressources liées à une barrière sont libérées avec **`pthread_barrier_destroy`**`()`
- Toutes ces fonctions renvoient 0 en cas de succès.

Exemple

```
#include <pthread.h>

pthread_barrier_t bar;

void* func() {
        printf("2nd thread before barrier\n");
        pthread_barrier_wait(&bar);
        printf("2nd thread after barrier \n");
}

int main() {
        printf("main thread started\n");
        pthread_barrier_init(&bar, NULL, 2);

        pthread_t thread;
        pthread_create(&thread, NULL, func, NULL);

        pthread_barrier_wait(&bar);
        printf("main thread finished\n");
        pthread_join(thread, NULL);

        pthread_barrier_destroy(&bar);
        return EXIT_SUCCESS;
}
```