



Programmation Parallèle et Distribuée

Cours 7 : Synchronisation entre threads

Patrick Carribault

David Dureau

Marc Pérache (marc.perache@cea.fr)



Introduction

- Programmation mémoire partagée
 - Utilisation de threads
 - Modèle fork/join
- Partage de variable
 - Gestion des lecture/écriture aux variables partagées
 - Nécessité de synchroniser ces accès



Producteur/consommateur

- Problème du producteur/consommateur :
 - le programme est constitué de deux threads ; le premier lit les caractères au clavier et le second se charge de les afficher.
- Notes
 - Le thread principal (le père) se charge de la création de ses fils et de l'attente de leur mort.
 - Cette disparition est programmée à l'arrivée du caractère "F".



Solution précédente

```
volatile char theChar = '\\0';
volatile char afficher = 0;

void *lire (void *name)
{
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}

void *affichage (void *name)
{
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```



Solution précédente

```
int main (void)
{
    pthread_t filsA, filsB;
    if (pthread_create (&filsA, NULL, affichage, "AA")) {
        perror ("pthread create");
        exit (EXIT_FAILURE);
    }
    if (pthread_create (&filsB, NULL, lire, "BB")) {
        perror ("pthread create");
        exit (EXIT_FAILURE);
    }
    if (pthread_join (filsA, NULL))
        perror ("pthread join");
    if (pthread_join (filsB, NULL))
        perror ("pthread join");
    printf ("Fin du pere\n");
    return (EXIT_SUCCESS);
}
```



Plan du cours 7

- Section critique
 - Exclusion mutuelle : Mutex
- Sémaphore
 - Compteur de ressources partagées
- Conditions
- Clés
 - Variables privées par thread
 - Notion de TLS



Exclusion mutuelle (mutex)



Mutex - Définition

- Définition:
 - un mutex est un objet d'exclusion mutuelle. Il sert à protéger des données partagées de modification concurrentes et permet d'implémenter des section critiques.
- Un mutex à deux états:
 - Déverrouillé (pris par aucun thread).
 - Verrouillé (appartenant à un thread).
- Un thread qui tente de verrouiller un mutex déjà verrouillé, est suspendu jusqu'à ce que le mutex soit déverrouillé



Mutex - Définition

- Intérêt :
 - Rendre la main à l'ordonnanceur.
 - Assurer l'équité
- Limitations :
 - Un mutex ne peut être pris que par un seul thread à la fois.
 - Il ne peut y avoir qu'un seul thread dans la section critique.
 - Seul le thread qui a verrouillé le mutex peut le déverrouiller



Mutex - Primitives

- Initialisation d'un mutex

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)`

- Initialisation statique d'un mutex

- `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`

- Destruction d'un mutex

- `int pthread_mutex_destroy(pthread_mutex_t *mutex)`



Mutex - Primitives

- Verrouillage d'un mutex

- `int pthread_mutex_lock(pthread_mutex_t *mutex)`

- Déverrouillage d'un mutex

- `int pthread_mutex_unlock(pthread_mutex_t *mutex)`

- Tentative de verrouillage

- `int pthread_mutex_trylock(pthread_mutex_t *mutex)`
 - La valeur de retour permet de déterminer si le mutex a été acquis ou non.



Mutex - Exemple

- Une variable globale partagée x peut être protégée par un mutex
 - Tous les accès et modifications de x doivent être entourés de paires d'appels à `pthread_mutex_lock()` et `pthread_mutex_unlock()`

- Exemple :

```
int x;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

/* ... */

pthread_mutex_lock(&mut);
/* opération sur x */
pthread_mutex_unlock(&mut);
```



Producteur/consommateur (rappel)

```
volatile char theChar = '\\0';
volatile char afficher = 0;

void *lire (void *name)
{
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}

void *affichage (void *name)
{
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```

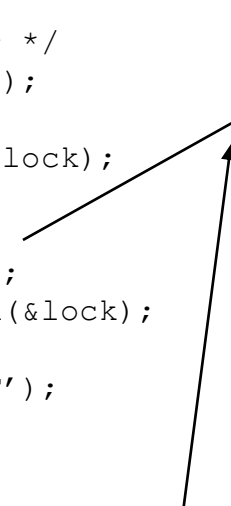


Producteur/consommateur

```
volatile char theChar = '\0';  
volatile char afficher = 0;  
pthread_mutex_t lock =  
    PTHREAD_MUTEX_INITIALIZER;
```

```
void *lire (void *name)  
{  
    do {  
        /* Attendre mon tour */  
        while (afficher == 1);  
  
        pthread_mutex_lock(&lock);  
        /* Donner le tour */  
        afficher = 1;  
        theChar = getchar ();  
        pthread_mutex_unlock(&lock);  
  
    } while (theChar != 'F');  
    return NULL;  
}
```

```
void *affichage (void *name)  
{  
    do {  
        /* Attendre mon tour */  
        while (afficher == 0);  
  
        pthread_mutex_lock(&lock);  
        printf ("car = %c\n",  
            theChar);  
        /* Donner le tour */  
        afficher = 0;  
        pthread_mutex_unlock(&lock);  
  
    } while (theChar != 'F');  
    return NULL;  
}
```



Déplacement de l'affectation de
afficher pour éviter l'attente
active sur l'écriture



Anatomie d'un mutex

- Structure d'un mutex ainsi que d'un slot pour gérer la liste des threads en attente

```
typedef struct slot_s {  
    thread_t *thread;  
    struct slot_s *next;  
} slot_t;
```

```
typedef struct {  
    /* Compteur de threads dans ou en attente de section critique */  
    volatile int nb_threads;  
    /* Liste des thread bloqués */  
    volatile slot_t *list_first;  
    volatile slot_t *list_last;  
    /* Spinlock pour verrouiller les accès aux données du mutex */  
    spinlock_t lock;  
} mutex_t;
```



Anatomie d'un mutex

- Fonction de verrouillage : lock

```
void mutex_lock (mutex_t * m)
{
    slot_t slot;
    spinlock (&(m->lock));
    if (m->nb_thread == 0) {
        m->nb_thread = 1;
        spinunlock (&(m->lock));
    } else {
        slot.thread = thread_self ();
        enqueue (&slot, m);
        thread_self ()->status = blocked;
        register_spinunlock (&(m->lock));
        yield ();
    }
}
```




Anatomie d'un mutex

- Fonction de déverrouillage : unlock

```
void mutex_unlock (mutex_t * m)
{
    slot_t *slot;
    spinlock (&(m->lock));
    if (m->list_first != NULL) {
        slot = dequeue (m);
        wake (slot->thread);
    } else {
        m->nb_thread = 0;
    }
    spinunlock (&(m->lock));
}
```



Anatomie d'un mutex

- Fonction de test : trylock

```
int mutex_trylock (mutex_t * m)
{
    slot_t slot;
    spinlock (&(m->lock));
    if (m->nb_thread == 0) {
        m->nb_thread = 1;
        spinunlock (&(m->lock));
    }
    return 0;
}
spinunlock (&(m->lock));
return 1;
}
```



Anatomie d'un mutex récursif

- Fonction de verrouillage : lock

```
void mutex_lock (mutex_t * m) {
    slot_t slot;
    spinlock (&(m->lock));
    if (m->nb_thread == 0) {
        m->nb_thread = 1;
        m->owner = thread_self ();
        spinunlock (&(m->lock));
    } else {
        if (m->owner == thread_self ()) {
            m->step++;
            spinunlock (&(m->lock));
        } else {
            slot.thread = thread_self ();
            enqueue (&slot, m);
            thread_self ()->status = blocked;
            register_spinunlock (&(m->lock));
            yield ();
        }
    }
}
```



Anatomie d'un mutex récursif

- Fonction de verrouillage : unlock

```
void mutex_unlock (mutex_t * m) {
    slot_t *slot;
    spinlock (&(m->lock));
    if (m->step == 1) {
        m->step--;
        if (m->list_first != NULL) {
            slot = dequeue (m);
            wake (slot->thread);
        } else {
            m->nb_thread = 0;
        }
    } else {
        m->step--;
    }
    spinunlock (&(m->lock));
}
```



Anatomie d'un mutex récursif

- Fonction de test : trylock

```
int mutex_trylock (mutex_t * m){
    slot_t slot;
    spinlock (&(m->lock));
    if (m->nb_thread == 0) {
        m->nb_thread = 1;
        m->owner = thread_self ();
        spinunlock (&(m->lock));
        return 0;
    } else {
        if (m->owner == thread_self ()) {
            m->step++;
            spinunlock (&(m->lock));
            return 0;
        }
    }
    spinunlock (&(m->lock));
    return 1;
}
```



Sémaphores



Sémaphore - Définition

- Définition :
 - les sémaphores sont des compteur pour des ressources partagées par plusieurs threads.
- Opérations sur les sémaphores
 - Incrémenter le compteur
 - Décrémenter le compteur
 - Si l'on essaye de décrémenter un compteur déjà à zéro, alors le thread est bloqué
- Pour utiliser les sémaphores, il faut utiliser l'include `semaphore.h`



Sémaphore - Fonctions

- Initialisation d'un sémaphore

- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
- Sémaphore non nommé
- L'argument `value` spécifie la valeur initiale du sémaphore.
- L'argument `pshared` indique si ce sémaphore sera partagé entre les threads d'un processus ou entre processus.

- Destruction d'un sémaphore

- `int sem_destroy(sem_t *sem);`



Sémaphore - Fonctions

- Décrémentation (verrouillage) un sémaphore
 - `int sem_wait(sem_t *sem);`
 - Si la valeur du sémaphore est plus grande que 0, la décrémentation s'effectue et la fonction revient immédiatement.
 - Si le sémaphore vaut zéro, l'appel bloquera jusqu'à ce que soit il devienne disponible d'effectuer la décrémentation (c'est-à-dire la valeur du sémaphore n'est plus nulle).



Sémaphore - Fonctions

- Test de verrouillage

- `int sem_trywait(sem_t *sem);`
- Si la décrémentation ne peut pas être effectuée immédiatement, l'appel renvoie une erreur
- La variable `errno` vaut `EAGAIN` (l'appel n'est alors pas bloquant)

- Incrémentation (déverrouillage) d'un sémaphore

- `int sem_post(sem_t *sem);`
- Si, à la suite de cet incrément, la valeur du sémaphore devient supérieure à zéro, un autre processus ou thread bloqué dans un appel `sem_wait` sera réveillé et procédera au verrouillage du sémaphore.



Producteur/consommateur (rappel)

```
volatile char theChar = '\\0';
volatile char afficher = 0;

void *lire (void *name) {
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}

void *affichage (void *name) {
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```



Producteur/consommateur

```
volatile char theChar = '\\0';  
sem_t lock_lire;  
sem_t lock_affiche;
```

```
void *lire (void *name) {  
    do {  
        sem_wait(&lock_lire);  
        theChar = getchar ();  
        sem_post(  
            &lock_affiche);  
    } while (theChar != 'F');  
    return NULL;  
}
```

```
void *affichage (void *name)  
{  
    do {  
        sem_wait(  
            &lock_affiche);  
        printf ("car = %c\\n",  
            theChar);  
        sem_post (&lock_lire);  
    } while (theChar != 'F');  
    return NULL;  
}
```



Conditions



Conditions

- Définition :
 - Une condition est un mécanisme de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition (un *prédicat*) sur des données partagées soit vérifiée.
- Opérations fondamentales sur les conditions :
 - Signaler la condition (quand le prédicat devient vrai)
 - Attendre la condition en suspendant l'exécution du thread jusqu'à ce qu'un autre thread signale la condition
- Une variable condition doit toujours être associée à un mutex, pour éviter une condition concurrente où un thread se prépare à attendre une condition et un autre thread signale la condition juste avant que le premier n'attende réellement.



Conditions

- Initialisation d'une variable condition
 - `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
 - Utilisation des attributs par défaut si `cond_attr` vaut `NULL`.
- Initialisation statique
 - Utilisation de la constante `PTHREAD_COND_INITIALIZER`.
- Relancer un des threads attendant une variable condition
 - `int pthread_cond_signal(pthread_cond_t *cond);`
 - S'il n'existe aucun thread répondant à ce critère, rien ne se produit.
 - Si plusieurs threads attendent sur `cond`, seul l'un d'entre eux sera relancé, mais il est impossible de savoir lequel.



Conditions

- Relancer tous les threads attendant sur une variable condition
 - `int pthread_cond_broadcast(pthread_cond_t *cond);`
 - Rien ne se passe s'il n'y a aucun thread attendant sur cond.
- Déverrouiller atomiquement le mutex et attendre qu'une variable condition soit signalée
 - `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
 - L'exécution du thread est suspendue et ne consomme pas de temps CPU jusqu'à ce que la variable condition soit signalée.
 - Le mutex doit être verrouillé par le thread appelant à l'entrée de `pthread_cond_wait()`.
 - Avant de rendre la main au thread appelant, `pthread_cond_wait()` reverrouille mutex.



Conditions

- Le déverrouillage du mutex et la suspension de l'exécution sur la variable condition sont liés atomiquement.
 - Si tous les threads verrouillent le mutex avant de signaler la condition, il est garanti que la condition ne peut être signalée (et donc ignorée) entre le moment où un thread verrouille le mutex et le moment où il attend sur la variable condition.
- Destruction d'une variable condition
 - `int pthread_cond_destroy(pthread_cond_t *cond);`
 - Libération des ressources qu'elle possède.
 - Aucun thread ne doit attendre sur la condition à l'entrée de `pthread_cond_destroy()`.



Producteur/consommateur (rappel)

```
volatile char theChar = '\\0';
volatile char afficher = 0;
void *lire (void *name) {
    do {
        while (afficher == 1); /* attendre mon tour */
        theChar = getchar ();
        afficher = 1; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
void *affichage (void *name) {
    do {
        while (afficher == 0); /* attendre */
        printf ("car = %c\\n", theChar);
        afficher = 0; /* donner le tour */
    } while (theChar != 'F');
    return NULL;
}
```



Producteur/consommateur

```
volatile char theChar = '\\0';
pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_lire =
    PTHREAD_COND_INITIALIZER;
pthread_cond_t cond =
    PTHREAD_COND_INITIALIZER;

void *lire (void *name){
    do {
        pthread_mutex_lock(&lock);
        theChar = getchar ();
        pthread_cond_signal(&cond);
        pthread_cond_wait(
            &cond_lire, &lock);
        pthread_mutex_unlock(&lock);
    } while (theChar != 'F');
    return NULL;
}
```

```
void *affichage (void *name){
    do {
        pthread_mutex_lock (&lock);
        pthread_cond_wait (
            &cond, &lock);
        printf ("car = %c\\n",
            theChar);
        pthread_cond_signal(
            &cond_lire);
        pthread_mutex_unlock(&lock);
    } while (theChar != 'F');
    return NULL;
}
```



Producteur/consommateur

```
volatile char theChar = '\\0';
pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_lire =
    PTHREAD_COND_INITIALIZER;
pthread_cond_t cond =
    PTHREAD_COND_INITIALIZER;

void *lire (void *name){
    do {
        pthread_mutex_lock(&lock);
        theChar = getchar ();
        pthread_cond_signal(&cond);
        pthread_cond_wait(
            &cond_lire, &lock);
        pthread_mutex_unlock(&lock);
    } while (theChar != 'F');
    return NULL;
}
```

Bug car si lire commence on a un deadlock

```
void *affichage (void *name){
    do {
        pthread_mutex_lock (&lock);
        pthread_cond_wait (
            &cond, &lock);
        printf ("car = %c\\n",
            theChar);
        pthread_cond_signal(
            &cond_lire);
        pthread_mutex_unlock(&lock);
    } while (theChar != 'F');
    return NULL;
}
```



Producteur/consommateur

```
volatile char theChar = '\\0';
pthread_mutex_t lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_lire =
    PTHREAD_COND_INITIALIZER;
pthread_cond_t cond =
    PTHREAD_COND_INITIALIZER;
volatile int affiche = 1;

void *lire (void *name){
    do {
        while(affiche == 1);
        pthread_mutex_lock(&lock);
        affiche = 1;
        theChar = getchar ();
        pthread_cond_signal(&cond);
        pthread_cond_wait(
            &cond_lire, &lock);
        pthread_mutex_unlock(&lock);
    } while (theChar != 'F');
    return NULL;
}
```

```
void *affichage (void *name){
    do {
        pthread_mutex_lock (&lock);
        affiche = 0;
        pthread_cond_wait (
            &cond, &lock);
        printf ("car = %c\\n",
            theChar);
        pthread_cond_signal(
            &cond_lire);
        pthread_mutex_unlock(&lock);
    } while (theChar != 'F');
    return NULL;
}
```



Clés



Clés

- Les programmes ont souvent besoin de variables globales ou statiques ayant différentes valeurs dans des threads différents.
 - Comme les threads partagent le même espace mémoire, cet objectif ne peut être réalisé avec les variables usuelles.
 - Les données spécifiques à un thread POSIX sont la réponse à ce problème.
- Chaque thread possède un segment mémoire privé, le TSD (Thread-Specific Data : Données Spécifiques au Thread).
 - Cette zone mémoire est indexée par des clés TSD.
 - La zone TSD associe des valeurs du type `void *` aux clés TSD.
 - Ces clés sont communes à tous les threads, mais la valeur associée à une clé donnée est différente dans chaque thread.



Clés

- Pour concrétiser ce formalisme :
 - les zones TSD peuvent être vues comme des tableaux de pointeurs void *,
 - les clés TSD comme des indices entiers pour ces tableaux, et
 - les valeurs des clés TSD comme les valeurs des entrées correspondantes dans le tableau du thread appelant.
- Quand un thread est créé, sa zone TSD associe initialement NULL à toutes les clés.



Clés

- Allocation d'une nouvelle clé TSD
 - `int pthread_key_create(pthread_key_t *clé, void (*destr_function) (void *));`
 - Cette clé est enregistrée à l'emplacement pointée par clé.
 - Il ne peut y avoir plus de `PTHREAD_KEYS_MAX` clés allouées à un instant donné.
 - La valeur initialement associée avec la clé renvoyée est `NULL` dans tous les threads en cours d'exécution.
- Le paramètre `destr_function`, si différent de `NULL`, spécifie une fonction de destruction associée à une clé.
 - Quand le thread se termine par `pthread_exit()` ou par une annulation, `destr_function` est appelée avec en argument les valeurs associées aux clés de ce thread.
 - La fonction `destr_function` n'est pas appelée si cette valeur est `NULL`.
 - L'ordre dans lequel les fonctions de destruction sont appelées lors de la fin du thread n'est pas spécifiée.



Clés

- Avant que la fonction de destruction soit appelée, la valeur NULL est associée à la clé dans le thread courant.
- Une fonction de destruction peut cependant réassocier une valeur différente de NULL à cette clé ou une autre clé.
- Pour gérer ce cas de figure, si après l'appel de tous les destructeurs pour les valeurs différentes de NULL, il existe toujours des valeurs différentes de NULL avec des destructeurs associés, alors la procédure est répétée.



Clés

- Désallocation d'une clé TSD
 - `int pthread_key_delete(pthread_key_t clé);`
 - Pas de vérification si des valeurs différentes de NULL sont associées avec cette clé dans les threads en cours d'exécution,
 - Pas d'appel à la fonction de destruction associée avec cette clé.
- Changement de la valeur associée avec une clé
 - `int pthread_setspecific(pthread_key_t clé, const void *pointer);`
- Accès à la valeur d'une clé
 - `void * pthread_getspecific(pthread_key_t clé);`
 - Valeur correspondant au thread appelant.



TLS

- TLS (thread local storage) :
 - Espace de stockage spécifique à chaque thread.
 - La différence avec les clés présentées auparavant, est que c'est ici le compilateur qui se charge du travail.
- Mot clé du langage : `__thread`
 - Exemple : `__thread int v;`
 - Permet de déclarer la variable `v` comme étant locale à chaque thread.
- Elle s'utilise ensuite (lecture/écriture) comme une variable normale.
 - Le compilateur génère certains appels de fonctions au moment de la génération de codes
 - Optimisation du *linker* pour diminuer le coût d'accès à ces variables