Concurrence : condition variables et sémaphores

Thibaud Martinez
thibaud.martinez@dauphine.psl.eu

Dauphine | PSL
UNIVERSITÉ PARIS

Cette présentation couvre les chapitres 30, 31 et 32 de Operating Systems: Three Easy Pieces.

Les diapositives sont librement adaptées de diapositives de *Youjip Won (Hanyang University)*.

Comment un thread parent peut vérifier qu'un thread enfant a fini de réaliser une action ?

```c
void *child(void *arg) {
    // Action à réaliser.
    // Comment indiquer au parent que l'action est réalisée ?
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    // Comment attendre que l'enfant ait réalisée l'action ?
    return 0;
}
```

Une approche basée sur l'attente active

```c
int done = 0;

void *child(void *arg) {
    // Action à réaliser.
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    while (done == 0)
        ; // spin
    return 0;
}
```

Cette méthode est inefficace : le parent **gaspille du temps CPU** en attendant l'enfant.

Condition variable

Il existe de nombreux cas où un thread souhaite **vérifier si une condition est vraie** avant de **poursuivre son exécution**.

La primitive de synchronisation ***condition variable*** permet cela, à travers deux opérations:

- **Attente de la condition** *(wait)* : les threads peuvent se placer dans une file d'attente lorsqu'un état d'exécution n'est pas atteint.
- **Signalisation de la condition** *(signal)* : un autre thread, lorsqu'il change d'état, peut réveiller l'un des threads en attente et lui permettre de continuer.

Utilisation des `pthread_cond_t`

Déclaration et initialisation

```c
#include <pthread.h>

pthread_cond_t c;                // Déclaration de la condition variable.
c = PTHREAD_COND_INITIALIZER;    // Une initialisation est nécessaire.
```

Opérations

```c
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

- L'appel `wait()` prend un mutex en paramètre.
- L'appel `wait()` libère le verrou et met le thread appelant en sommeil.
- Lorsque le thread se réveille, il doit réacquérir le verrou.

Utilisation d'une condition variable pour que le parent attende l'enfant

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

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    thr_exit();
    return NULL;
}
```

```c
void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    return 0;
}
```

L'importance de la variable d'état `done`

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Imaginons le cas où l'enfant s'exécute immédiatement.

- L'enfant émettra un signal, mais il n'y a pas de thread endormi attendant la condition.
- Lorsque le parent s'exécute, il appelle wait et reste bloqué, aucun thread ne le réveillera jamais.

L'importance du mutex

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {
    while (done == 0)
        Pthread_cond_wait(&c);
}
```

Il s'agit ici d'une *race condition* subtile.

- Le parent appelle `thr_join()`.
- Le parent vérifie la valeur de `done`. Celle-ci est de `0` et le parent essaie de s'endormir.
- Juste avant d'appeler `wait()` pour s'endormir, le parent est interrompu et l'enfant s'exécute.
- L'enfant change la variable d'état `done` en `1` et émet un signal. Mais aucun thread n'est en attente et donc aucun thread n'est réveillé.
- Lorsque le parent s'exécute à nouveau, il s'endort pour toujours.

Sémaphore

Il s'agit d'une primitive de synchronisation contentant une **valeur entière**.

→ *inventé par Edsger Dijkstra.*

La valeur du sémaphore est manipulée grâce à deux opérations : `sem_wait()` et `sem_post()`.

Initialisation

```c
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialise s à la valeur 1
```

→ Le deuxième argument, `0`, indique que le sémaphore est partagé entre les threads d'un même processus.

Opérations

```
int sem_wait(sem_t *s)
```

Décrémente la valeur du sémaphore `s` de `1` et met le thread en attente si la valeur du sémaphore `s` est négative.

→ Lorsqu'elle est négative, la valeur du sémaphore est égale au **nombre de threads en attente**.

```
int sem_post(sem_t *s)
```

Incrémente la valeur du sémaphore `s` de `1`. S'il y a un ou plusieurs threads en attente, un des threads est réveillé.

À quelle valeur le sémaphore doit-il être initialisé pour être utilisé comme un verrou ?

```
sem_t m;
sem_init(&m, 0, X); // quelle devrait être la valeur de X ?
```

Sémaphores binaires (verrous)

Un sémaphore initialisé avec une valeur de `1` se comporte comme un verrou.

```
sem_t m;
sem_init(&m, 0, 1);

sem_wait(&m);
// section critique
sem_post(&m);
```

Deux threads utilisant un sémaphore binaire

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` **returns**	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` **returns**	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` **returns**	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` **returns**	Run

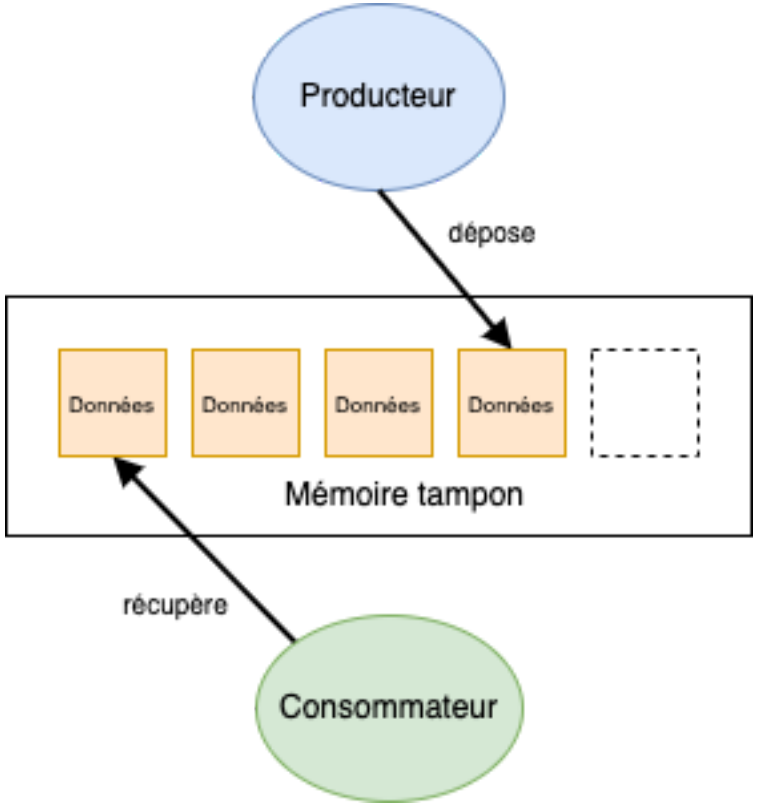
Le problème du producteur / consommateur (*bounded buffer*)

Producteur

- Produit des éléments de données.
- Souhaite placer ces données dans une mémoire tampon.

Consommateur

- Récupère les données dans la mémoire tampon et les consomme d'une manière ou d'une autre.



Exemple : Serveur web multithreads

- Un producteur place les requêtes HTTP dans une file d'attente.
- Les consommateurs extraient les requêtes de cette file d'attente et les traitent.

Contraintes

- Le **producteur** doit attendre qu'il y ait des emplacements **disponibles** dans la mémoire tampon pour les remplir.
- Le **consommateur** doit attendre que des éléments de données aient été **placés** dans le tampon pour les consommer.

```
int buffer[MAX];
int fill = 0;
int use = 0;

// Utilisé par le producteur pour placer les données dans le tampon.
void put(int value) {
    buffer[fill] = value;       // ligne f1
    fill = (fill + 1) % MAX;    // ligne f2
}

// Utilisé par le consommateur pour récupérer les données du tampon.
int get() {
    int tmp = buffer[use];      // ligne g1
    use = (use + 1) % MAX;      // ligne g2
    return tmp;
}
```

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);       // ligne p1
        put(i);                 // ligne p2
        sem_post(&full);        // ligne p3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);        // ligne c1
        tmp = get();            // ligne c2
        sem_post(&empty);       // ligne c3
    }
}
```

```c
int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers sont vides au début
    sem_init(&full, 0, 0);    // … et 0 sont pleins
    // ...
}
```

Imaginons que `MAX` soit supérieur à `1`.

- S'il y a plusieurs producteurs, une *race condition* peut se produire à la ligne f1.
- Cela signifie que les anciennes données sont **écrasées**.

Nous avons oublié l'**exclusion mutuelle** !

→ Le remplissage d'un tampon et l'incrémentation de l'index dans le tampon constitue une section critique.

Une solution : mettre en œuvre l'exclusion mutuelle

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);   // ligne p0 (NOUVELLE LIGNE)
        sem_wait(&empty);   // ligne p1
        put(i);             // ligne p2
        sem_post(&full);    // ligne p3
        sem_post(&mutex);   // ligne p4 (NOUVELLE LIGNE)
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);   // ligne c0 (NOUVELLE LIGNE)
        sem_wait(&full);    // ligne c1
        int tmp = get();    // ligne c2
        sem_post(&empty);   // ligne c3
        sem_post(&mutex);   // ligne c4 (NOUVELLE LIGNE)
        printf("%d\n", tmp);
    }
}
```

Imaginons deux threads : un producteur et un consommateur.

- Le consommateur acquiert le mutex (ligne c0).
- Le consommateur appelle `sem_wait()` sur le sémaphore `full` (ligne c1).
- Le consommateur est bloqué et cède le processur.
- Le consommateur détient toujours le mutex !
- Le producteur appelle `sem_wait()` sur le sémaphore binaire du mutex (ligne p0).
- Le producteur est maintenant bloqué et attend lui aussi, ce qui constitue un interblocage classique (*deadlock*).

Finalement, une solution qui fonctionne

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);   // ligne p1
        sem_wait(&mutex);   // ligne p1.5 (MUTEX DÉPLACÉ ICI...)
        put(i);             // ligne p2
        sem_post(&mutex);   // ligne p2.5 (... ET ICI)
        sem_post(&full);    // ligne p3
    }
}
```

```c
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);         // ligne c1
        sem_wait(&mutex);        // ligne c1.5 (MUTEX DÉPLACÉ ICI...)
        int tmp = get();         // ligne c2
        sem_post(&mutex);        // ligne c2.5 (... ET ICI)
        sem_post(&empty);        // ligne c3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
        // ...
        sem_init(&empty, 0, MAX);
        sem_init(&full, 0, 0);
        sem_init(&mutex, 0, 1);   // mutex=1 car c'est un verrou
        // ...
}
```

Comment implémenter un sémaphore ?

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// un seul thread peut appeler cette fonction
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}
```

```
void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Contrairement au sémaphores purs tels que définis par Dijkstra, la valeur de notre Zemaphore ne sera **jamais inférieure à `0`**, et donc ne reflètera pas le **nombre de threads en attente** lorsqu'elle est négative.

→ Ce comportement est plus facile à mettre en œuvre et correspond à l'implémentation actuelle de Linux.

Problèmes de concurrence communs

Les problèmes de concurrence sont souvent **sources de bugs** dans les programmes.

Une étude de *Lu et al.* analyse 4 applications *open-source* populaires pour identifier quels sont les bugs qu'on rencontre le plus couramment.

Application	Type d'application	Autre que deadlocks	Deadlocks
MYSQL	Base de données	14	9
Apache	Serveur web	13	4
Firefox	Navigateur web	41	16
Open Office	Suite bureautique	6	2
Total		**74**	**31**

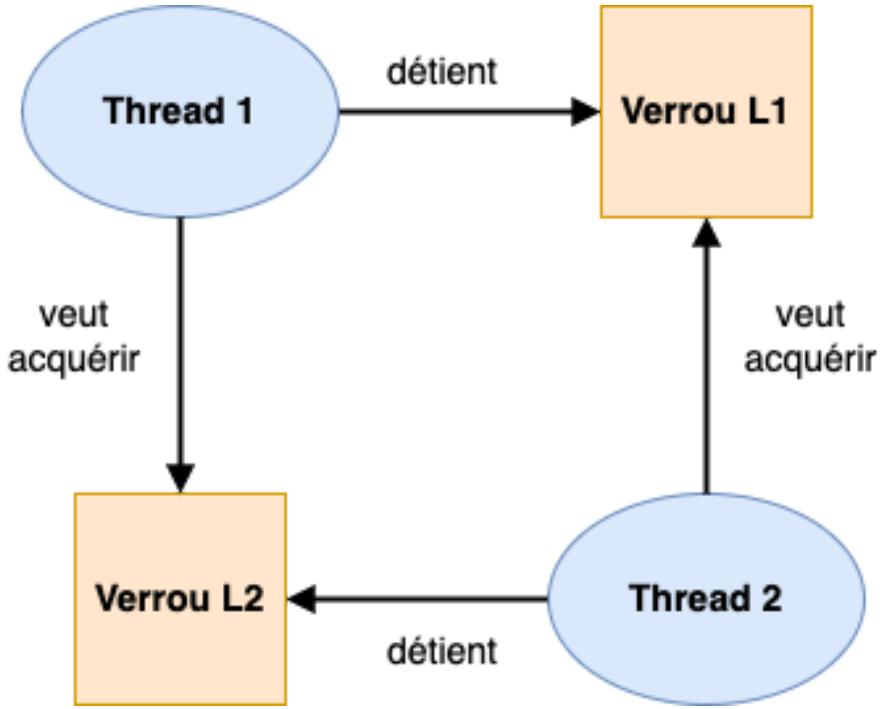
Lu, S., Park, S., Seo, E. and Zhou, Y., 2008, March. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. ASPLOS '08, March 2008, Seattle, Washington.

Interblocages (*deadlocks*)

```
Thread 1:       Thread 2:
lock(L1);       lock(L2);
lock(L2);       lock(L1);
```

Présence d'un cycle :

- Thread 1 détient le verrou L1 et en attend un autre, L2.
- Thread 2 détient le verrou L2 et attend que L1 soit libéré.



Bugs dus à la violation de l'atomicité

Une région de code est censée être atomique, mais l'atomicité n'est pas respectée pendant l'exécution.

Exemple simple trouvé dans MySQL : deux threads différents accèdent au champ `proc_info` dans la structure `thd`.

```
// Thread 1
if (thd->proc_info){
    ...
    fputs(thd->proc_info, ...);
    ...
}

// Thread 2
thd->proc_info = NULL;
```

Solution : ajouter des **verrous** autour des références aux variables partagées.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;


// Thread 1
pthread_mutex_lock(&lock);
if (thd->proc_info){
    ...
    fputs(thd->proc_info, ...);
    ...
}
pthread_mutex_unlock(&lock);

// Thread 2
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

Bugs dus au non-respect de l'ordre d'exécution

L'ordre souhaité entre deux accès à la mémoire est inversé.

→ **A** devrait toujours être exécuté avant **B**, mais l'ordre n'est pas respecté pendant l'exécution.

Exemple : le code de Thread 2 suppose que la variable `mThread` a déjà été initialisée (et n'est pas `NULL`).

```
// Thread 1
void init(){
    mThread = PR_CreateThread(mMain, ...);
}

// Thread 2
void mMain(…){
    mState = mThread->State
}
```

Solution : garantir l'ordre d'exécution en utilisant des **conditions variables**.

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0;

// Thread 1
void init(){
    mThread = PR_CreateThread(mMain,…);

    // signale que le thread a été créé.
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
}
```

```
// Thread 2
void mMain(…){
    // attendre que mThread soit initialisé
    pthread_mutex_lock(&mtLock);
    while(mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

    mState = mThread->State;
}
```

Pour aller plus loin

- The Little Book of Semaphores, Allen B. Downey
- Dijkstra, E.W., 2002. Cooperating sequential processes