Concurrence : threads et verrous

Thibaud Martinez
thibaud.martinez@dauphine.psl.eu

Dauphine | PSL
UNIVERSITÉ PARIS

Cette présentation couvre les chapitres 26 et 28 de Operating Systems: Three Easy Pieces.

Les diapositives sont librement adaptées de diapositives de *Youjip Won (Hanyang University)* et *Mathias Payer (EPFL)*.

Thread

- Une nouvelle abstraction pour un programme en cours d'exécution.
- Un "processus léger".
- Un programme multi-thread possède plusieurs threads lors de son exécution.
- Les threads d'un même processus **se partagent son espace d'adressage**.

Commutation de contexte entre threads

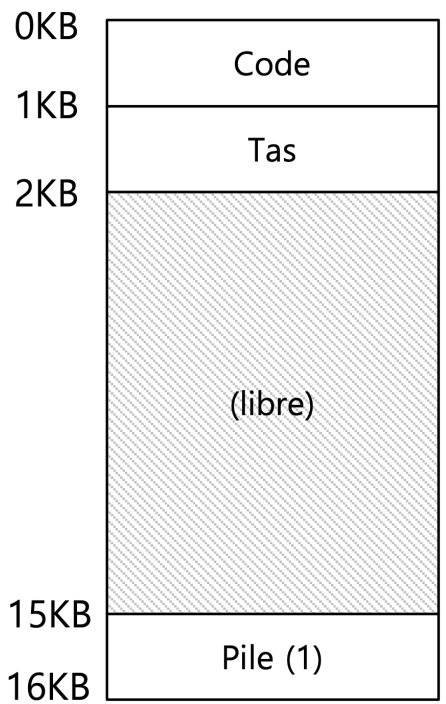
Chaque thread possède son propre **pointeur d'instruction (*program counter*)** et ses propres **registres**.

Lors du passage d'un thread (T1) à l'autre (T2),

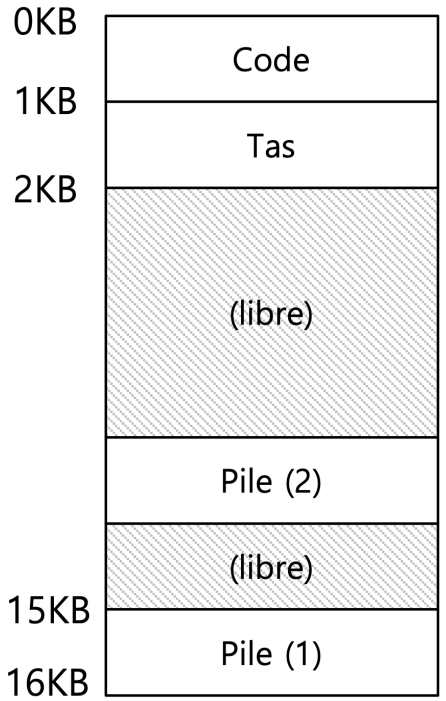
- L'état des registres de T1 est sauvegardé.
- L'état des registres de T2 est restauré.
- **L'espace d'adressage reste le même**.

Pile de thread

Il y a **une pile (*stack*) par thread**.



L'espace d'adressage d'un programme mono-thread



L'espace d'adressage d'un programme avec deux threads

Pourquoi utiliser des threads ?

1. **Parallélisation** : diviser le travail entre plusieurs processeurs.
2. Éviter de bloquer la progression d'un programme à cause d'**entrées/sorties** lentes.

Note: cela peut être fait en utilisant des processus, mais, étant donné que les threads d'un programme partagent le même espace d'adressage, il est facile de partager des données entre eux.

Parallélisme et concurrence

Parallélisme : plusieurs threads (ou processus) travaillent sur une même tâche en utilisant plusieurs processeurs simultanément.

Concurrence : les tâches peuvent démarrer, s'exécuter et se terminer dans des périodes de temps qui se chevauchent, par exemple par multiplexage temporel en intercalant leurs exécutions, ou par parallélisme lorsqu'elles sont exécutées en même temps.

Accès concurrent (*race condition*)

```c
int counter = 0;

void * mythread(void *arg) {
    printf("%s starts\n", (char*) arg);
    for (int i=0; i < 1000000; ++i) {
        counter = counter + 1;
    }
    printf("%s done\n", (char*) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, mythread, "T1");
    pthread_create(&t2, NULL, mythread, "T2");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter: %d (expected: %d)\n", counter, 1000000*2); return 0;
}
```

Que se passe-t-il ?

```
$ ./race
T1 starts
T2 starts
T1 is done
T2 is done
Counter: 1150897 (expected: 2000000)
```

En plus d'obtenir un résultat **erroné**, le résultat est **non-déterministe** et dépend de l'ordonnancement des threads.

`mythread` en langage d'assembleur :

```
mov 0x8049a1c, %eax      ; load value
add $0x1, %eax           ; increment
mov %eax, 0x8049a1c      ; increment
```

OS	Thread1	Thread2	(après l'instruction) %eax	counter
	`mov 0x8049a1c, %eax`		50	50
	`add $0x1, %eax`		51	50
`interruption`				
`sauvegarde état T1`				
`restaure état T2`				
		`mov 0x8049a1c, %eax`	0	50
		`add $0x1, %eax`	50	50
		`mov %eax, 0x8049a1c`	51	50
			51	51
`interruption`				
`sauvegarde état T2`				
`restaure état T1`			51	50
	`mov %eax, 0x8049a1c`		51	**51**

On s'attend à un résultat de **52**.

Les deux threads chargent la même valeur, l'incrémentent et la réécrivent. L'ajout d'un thread est perdu !

Section critique

Une portion de code qui accède à une variable partagée et qui ne doit pas être exécutée simultanément par plus d'un thread.

Exemple : `counter = counter + 1`

→ L'exécution d'une section critique par plusieurs threads peut entraîner des accès concurrents.
→ Nécessité de supporter l'**atomicité** pour les sections critiques : les instructions sont toutes exécutées d'un coup où ne le sont pas.

Qu'est-ce qu'un verrou (*lock*) ?

Primitive de synchronisation qui veille à ce qu'une section critique s'exécute comme s'il s'agissait d'une instruction atomique unique.

- *Exemple :* mise à jour d'une variable partagée

```
counter = counter + 1;
```

- Ajout de code autour de la section critique.

```
lock_t mutex;

lock(&mutex);
counter = counter + 1;
unlock(&mutex);
```

Exclusion mutuelle

Exigence selon laquelle un thread n'entre jamais dans une section critique alors qu'un thread concurrent accède déjà à ladite section critique.

Un ***mutex (mutual exclusion object)*** est une variable qui garantie l'exclusion mutuelle.

→ *mutex* = verrou

État du verrou

La variable lock (ici nommée `mutex`) contient l'état du verrou.

- **disponible** (ou déverrouillé ou libre) : aucun thread ne détient le verrou.
- **acquis** (ou verrouillé ou détenu) : un seul thread détient le verrou et se trouve vraisemblablement dans une section critique.

La sémantique de `lock()`

La fonction `lock()`:

- Essaie d'acquérir le verrou.
- Si aucun autre thread ne détient le verrou, le thread acquiert le verrou.
- À l'entrée dans la section critique, ce thread est considéré comme le propriétaire du verrou.
- Les autres threads ne peuvent pas entrer dans la section critique tant que le premier thread qui détient le verrou s'y trouve.

Pthread mutex

Le nom donné aux verrous par POSIX (famille de normes techniques).

Utilisé pour garantir l'exclusion mutuelle entre threads.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock);
counter = counter + 1;
Pthread_mutex_unlock(&lock);
```

Note : nous pouvons utiliser différents verrous pour protéger différentes variables afin d'améliorer les performances.

pthread_mutex_lock(3p) — Linux manual page

Comment construire un verrou ?

- Comment construire un verrou efficace ? Les verrous efficaces permettent l'exclusion mutuelle à faible coût en termes de performance.
- Quel support matériel est nécessaire ?
- Quel est le support du système d'exploitation ?

Évaluation de l'implémentation des verrous

- **Exclusion mutuelle** : Le verrou fonctionne-t-il, empêchant plusieurs threads d'entrer dans une section critique ?
- **Equité** : Chaque thread en lice pour le verrou a-t-il une chance équitable de l'acquérir (pas de famine) ?
- **Performance** : Les surcharges temporelles ajoutées par l'utilisation du verrou sont-elles acceptables ?

Une solution simple : contrôler les interruptions

Désactiver les interruptions pour les sections critiques.

```
void lock() { DisableInterrupts() ; }
void unlock() { EnableInterrupts() ; }
```

L'une des premières solutions utilisées pour fournir une exclusion mutuelle.

Problèmes :

- Place trop de confiance dans les applications : un programme malveillant pourrait monopoliser le processeur.
- Ne fonctionne pas avec plusieurs processeurs.
- Le code qui masque ou démasque les interruptions est exécuté lentement par les processeurs modernes.

Une tentative incorrecte

Utiliser une variable drapeau (`flag`) pour indiquer si le verrou est détenu ou non.

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *mutex) {
        // 0 -> lock is available, 1 -> held
        mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
                ;  // spin-wait (do nothing)
        mutex->flag = 1;  // now SET it !
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Problème 1 : Pas d'exclusion mutuelle (supposer que `flag=0` pour commencer).

Thread1	Thread2
appelle `lock()`	
`while (flag == 1)`	
commutation de contexte	
	appelle `lock()`
	`while (flag == 1)`
	`flag = 1;`
	commutation de contexte
`flag = 1;` // flag à 1 aussi !	

Problème 2 : L'**attente active (*"spin-waiting"*)** fait perdre du temps à l'attente d'un autre thread.

Nous avons besoin d'une instruction atomique supportée par le matériel !

Instruction *test-and-set (Atomic exchange)*

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr; // récupère l'ancienne valeur dans 'ptr'
    *ptr = new;     // sauvegarde 'new' dans 'ptr'
    return old;     // renvoie l'ancienne valeur
}
```

Cette séquence d'opérations est exécutée **de façon atomique**.

Un simple *spin lock* en utilisant *test-and-set*

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
            ;   // spin-wait
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Note : on suppose un ordonnanceur préemptif.

Évaluation des spin locks

- **Correct** : oui
  Le spin lock ne permet qu'à un seul thread d'entrer dans la section critique.
- **Équité** : non
  Les spin locks ne fournissent aucune garantie d'équité.
  En effet, un thread en attente active peut attendre indéfiniment.
- **Performance** :
  Dans le cas d'un seul processeur, les surcoûts liés aux performances peuvent être importantes.

Instruction *compare-and-swap*

```
int CompareAndSwap(int *ptr, int expected, int new) {
        int actual = *ptr;
        if (actual == expected) { *ptr = new; }
        return actual;
}
```

- Test si la valeur à l'adresse (`ptr`) est égale à la valeur attendue.
- Si c'est le cas, met à jour l'emplacement mémoire pointé par ptr avec la nouvelle valeur.
- Dans les deux cas, renvoie la valeur réelle à cet emplacement mémoire.

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
            ; // spin
}
```

Instruction fetch-and-add

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;     // incrémente la valeur
    return old;         // renvoie la valeur avant l'incrémentation
}
```

Ticket lock avec *fetch-and-add*

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
            ; // spin
}

void unlock(lock_t *lock) { FetchAndAdd(&lock->turn); }
```

Assure la progression de tous les threads → **équité**.

Éviter l'attente active

- Les *spin lock* basés sur le matériel sont simples et fonctionnent.
- Cependant, dans certains cas, ces solutions peuvent être très **inefficaces**.
- Chaque fois qu'un thread est en attente active, il perd un intervalle de temps entier à ne rien faire d'autre que de vérifier une valeur.

Comment éviter l'attente active ? Nous avons besoin du système d'exploitation.

Une approche simple : *just yield*

Lorsque qu'un thread s'apprête à attendre, il confie le processeur à un autre thread.

→ `yield()` fait passer l'appelant de l'état *running* à l'état *ready*.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Problème : le coût d'une commutation de contexte peut être substantiel et le problème de la famine existe toujours.

Utiliser des files : *"sleeping instead of spinning"*

- **File d'attente** : garde une trace des threads qui attendent d'acquérir le verrou.
- `park()` : met en sommeil un thread appelant
- `unpark(threadID)` : réveille un thread désigné par `threadID`.

```
typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;

void lock_init(lock_t *m) {
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Futex

Linux fournit un futex (similaire aux fonctions `park` et `unpark`).

- `futex_wait(adress, expected)` : met en veille le thread appelant.
  Si la valeur à l'adresse n'est pas égale à la valeur attendue, l'appel retourne immédiatement.
- `futex_wake(adresse)` : réveille un thread qui attend dans la file d'attente.

futex(2) — Linux manual page

Verrouillage en deux phases

Constat : l'attente active peut être pertinente si le verrou est sur le point d'être libéré.

- **Première phase**
  Le verrou attent activement pendant un certain temps, dans l'espoir d'acquérir le verrou. Si le verrou n'est pas acquis au cours de cette première phase, une deuxième phase est entamée.
- **Deuxième phase**
  L'appelant est endormi. Il n'est réveillé que si le verrou se libère ultérieurement.

Pourquoi étudier ces sujets dans le cours de systèmes d'exploitation ?

Le système d'exploitation a été le premier programme concurrent, et de nombreuses techniques ont été créées pour être utilisées au sein du système d'exploitation.

Plus tard, avec les processus multithreads, les programmeurs d'applications ont également dû tenir compte de ces éléments.

Concepts clés

- Une **section critique** est une portion de code qui accède à une ressource partagée.
- Une **accès concurrent (*race condition*)** se produit si plusieurs threads entrent dans la section critique à peu près en même temps ; tous deux tentent de mettre à jour la ressource partagée.
- Pour éviter les problèmes liès aux accès concurrents, les threads doivent utiliser des **primitives de synchronisation** comme les verrous.
- Une opération **atomique** s'exécute entièrement sans pouvoir être interrompue avant la fin de son déroulement.

Pour aller plus loin

- Concurrency is not parallelism, Rob Pike
- Dijkstra, E.W., 2002. Cooperating sequential processes