Concurrence

M2 SETI B4 / MS SE SE758

Guillaume Duc
guillaume.duc@telecom-paris.fr
2020–2021

Plan

Introduction

Variables atomiques

Mutex

Spinlock

Sémaphores

Conclusion

Concurrence ?

- Plusieurs parties de votre pilote peuvent s'exécuter en même temps ou être interrompues l'une par l'autre
  - Pendant l'exécution d'une fonction, une interruption en provenance de votre périphérique peut survenir et donc votre gestionnaire d'interruption peut s'exécuter au milieu de l'exécution d'une autre fonction
  - Sur un système multi-processeurs, deux fonctions de votre pilote (voir la même fonction) peuvent s'exécuter en même temps sur deux processeurs différents

Problème

- Il est donc important de contrôler l'accès aux différentes ressources de votre pilote
- Cela peut concerner les structures de données pour lesquels les modifications doivent être atomiques (on ne doit jamais voir une structure dans un état en cours de modification)
- Mais cela concerne également l'accès au périphérique lui-même, par exemple si une transaction nécessite plusieurs échanges ininterrompus avec le périphérique

Principales primitives

- Le noyau offre des primitives de synchronisation permettant, par exemple, de réaliser de l'*exclusion mutuelle* (c'est-à-dire s'assurer qu'à un instant donné, au plus un seul flot d'exécution est dans une *section critique*)
  - Variables atomiques
  - *Mutex*
  - *Spinlock*
  - Sémaphores

Plan

Introduction

Variables atomiques

Mutex

Spinlock

Sémaphores

Conclusion

Principe

- Sur certaines architectures, la simple incrémentation d'un entier n'est pas une opération atomique (un autre processeur essayant de lire la valeur de cet entier pendant l'opération peut récupérer une valeur autre que l'ancienne ou la nouvelle)
- Le noyau fournit le type `atomic_t` défini dans `<linux/types.h>` qui contient un entier signé (dont au minimum 24 bits sont utilisables)

Opérations sur les variables atomiques

<asm/atomic.h>

```
/* Lecture / ecriture */
void atomic_set(atomic_t *v, int v);
int atomic_read(atomic_t *v);

/* Operations arithmetiques */
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);

int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);

/* Renvoient une valeur != 0 si le resultat de l'operation est nul */
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

Opérations atomiques de manipulation de bits

<linux/bitops.h>

```
void set_bit(int nr, volatile unsigned long *addr);
void clear_bit(int nr, volatile unsigned long *addr);
void change_bit(int nr, volatile unsigned long *addr);

/* Modifient le bit et renvoient son _ancienne_ valeur */
int test_and_set_bit(int nr, volatile unsigned long *addr);
int test_and_clear_bit(int nr, volatile unsigned long *addr);
int test_and_change_bit(int nr, volatile unsigned long *addr);

int test_bit(int nr, const volatile unsigned long *addr);
```

- Documentation :
  `Documentation/core-api/atomic_ops.rst`

Plan

Introduction

Variables atomiques

Mutex

Spinlock

Sémaphores

Conclusion

Principe

- Un *mutex* est une primitive de synchronisation permettant de faire de l'*exclusion mutuelle*
- Un mutex ne peut être que dans deux états : libre (déverrouillé) ou pris (verrouillé)
- Deux opérations sont disponibles
  - *Verrouiller* : si le mutex est libre, il passe à l'état verrouillé ; si le mutex est verrouillé, le processus est *mis en sommeil* jusqu'à ce que le mutex soit libre (puis il est verrouillé)
  - *Déverrouiller* : le mutex passe à l'état libre (si un processus est en attente sur ce mutex, il est réveillé)
- Les fonctions relatives aux mutex sont déclarées dans `linux/mutex.h`

Initialisation

- Un mutex est représenté par la structure `struct mutex`
- Cette structure peut être initialisée
  - Statiquement avec la macro `DEFINE_MUTEX(mutex);`
  - Dynamiquement avec la fonction `void mutex_init(struct mutex *lock);`

Verrouillage

```
void mutex_lock(struct mutex *lock);
int mutex_lock_killable(struct mutex *lock);
int mutex_lock_interruptible(struct mutex *lock);
```

- Ces fonctions demandent à verrouiller (prendre) un mutex
- Si le mutex est disponible, il est verrouillé et ces fonctions retournent immédiatement
- Si le mutex est déjà verrouillé, elles *bloquent* (la tâche courante est mise en sommeil) jusqu'à ce que le mutex soit de nouveau disponible puis le prennent et rendent la main
- **Ces fonctions ne peuvent donc pas être utilisées dans un contexte où il est interdit de bloquer (top half par exemple)**

Verrouillage

```
void mutex_lock(struct mutex *lock);
int mutex_lock_killable(struct mutex *lock);
int mutex_lock_interruptible(struct mutex *lock);
```

- Pour la première fonction, l'attente ne peut pas être interrompue par un signal (avant que le mutex soit libre)
- Pour la seconde, l'attente peut être interrompue par un signal *SIGKILL*
- Pour la dernière, l'attente peut être interrompue par n'importe quel signal
- Dans ces deux derniers cas, si l'attente est interrompue par un signal, la fonction renvoie `-EINTR` et le *mutex n'est pas pris*

Verrouillage

```
int mutex_trylock(struct mutex *lock);
```

- Cette fonction essaie de verrouiller le mutex
  - S'il est disponible, elle le prend et retourne `1`
  - S'il n'est pas disponible, elle renvoie `0` immédiatement sans bloquer (et sans le verrouiller)

Déverrouillage

```
void mutex_unlock(struct mutex *lock);
```

- Cette fonction libère un mutex (qui doit avoir été pris par le thread courant)
- Si d'autres threads en attente sur l'opération de verrouillage, un sera réveillé et prendra le mutex

Plan

Introduction

Variables atomiques

Mutex

Spinlock

Sémaphores

Conclusion

Principe

- Un *spinlock* ressemble beaucoup à un *mutex*
- La grande différence entre les deux est que, si un spinlock est verrouillé, et qu'un autre thread essaie de le prendre, cet autre thread n'est pas mis en sommeil mais attend *activement* (en bouclant et en consommant du temps processeur)
- Un spinlock peut donc être utilisé dans un contexte où la tâche courante n'a pas le droit de bloquer
- Un spinlock doit être verrouillé le moins longtemps possible (quelques instructions au plus)
  - Une tâche ayant verrouillé un spinlock n'a pas le droit d'être mis en sommeil pendant ce temps
- Les fonctions relatives aux mutex sont déclarées dans `linux/spinlock.h`

Initialisation

- Un spinlock est représenté par le type `spinlock_t`
- Il peut être initialisé
  - Statiquement avec la macro `DEFINE_SPINLOCK(sl);`
  - Dynamiquement avec la fonction `void spin_lock_init(spinlock_t *lock);`

Opérations

```
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);
int spin_trylock(spinlock_t *lock);
```

- La première fonction essaie de verrouiller le spinlock
  - S'il n'est pas libre, elle va attendre activement jusqu'à ce qu'il le devienne
- La seconde fonction déverrouille le spinlock
- La dernière tente de verrouiller le spinlock (renvoie `0` sans attendre si le spinlock n'est pas libre)

Spinlocks et interruptions

- Un spinlock peut être utilisé pour protéger une section critique, uniquement dans le contexte d'un processus (*i.e.* entre deux fonctions autres que des gestionnaires d'interruptions dans votre pilote)
- Si une fonction de votre pilote qui a verrouillé un spinlock est interrompue, et que votre gestionnaire d'interruption essaie de verrouiller le même spinlock, il n'y arrivera pas et restera bloqué en attente active (si les deux fonctions s'exécutent sur le même processeur)

Verrouillage vis-à-vis des interruptions

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
int spin_trylock_irqsave(spinlock_t *lock, unsigned long flags);
```

- La première fonction verrouille le spinlock et désactive les interruptions sur le processeur courant
- La deuxième rend le spinlock et réactive les interruptions (si elles l'étaient lors de l'appel à `spin_lock_irqsave`)
- `spin_lock_irqsave` sauvegarde l'état des interruptions dans `flags` pour qu'il puisse être restauré dans la fonction `spin_unlock_irqrestore`

Spinlocks et interruptions (suite)

- La fonction `spin_lock_irqsave` peut donc être utilisée pour protéger une section de code critique quel que soit le contexte
- Pour reprendre l'exemple précédent, une fonction qui a verrouillé un spinlock à l'aide de cette méthode n'a pas à craindre d'être interrompue par une interruption puisque celles-ci sont désactivées sur le processeur courant
- Elle est donc certaine de pouvoir arriver jusqu'au point de déverrouillage de façon atomique
- Un gestionnaire d'interruption peut s'exécuter sur un autre processeur au même moment et s'il a besoin du spinlock il attendra jusqu'à sa libération qui arrivera forcément rapidement

Verrouillage vis-à-vis des *softirq* & *tasklets*

```
void spin_lock_bh(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

- Ces variantes se contente de désactiver/réactiver les *softirq* et les *tasklets* sur le processeur courant
- Elles ont donc un impact moins important que les fonctions précédentes (qui désactivaient toutes les interruptions)

Reader/writer spinlock

```
DEFINE_RWLOCK(lock);
/* ou */
rwlock_t lock;
rwlock_init(&lock);

/* Version lecteurs : tant qu'il n'y a que des lecteurs, ne bloquent
pas, bloquent seulement si un ecrivain a verrouille le spinlock */
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqsave(rwlock_t *lock, unsigned long flags);
void read_unlock_bh(rwlock_t *lock);

/* Version ecrivain : bloquent tant que des lecteurs ou un autre
ecrivain ont verrouille le spinlock */
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_bh(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqsave(rwlock_t *lock, unsigned long flags);
void write_unlock_bh(rwlock_t *lock);
```

Plan

Introduction

Variables atomiques

Mutex

Spinlock

Sémaphores

Conclusion

Principe

- Un *sémaphore* est une autre primitive permettant de gérer l'accès concurrent à une ressource
- Il est basé sur un compteur
- Deux opérations sont possibles
  - `up` : incrémente le compteur
  - `down` : si le compteur est strictement positif, il est décrémenté et s'il est égal à zéro, l'opération bloque en mettant la tâche en sommeil jusqu'à ce qu'il soit de nouveau strictement positif puis il est décrémenté
- Ces opérations peuvent être réalisées par des threads différents (un thread peut se contenter d'appeler `down` sans jamais appeler `up` par exemple)
- Les fonctions relatives aux sémaphores sont déclarées dans `linux/semaphore.h`

Initialisation

- Un sémaphore est représenté par la structure `struct semaphore`
- Cette structure peut être initialisée
  - Statiquement avec la macro `DEFINE_SEMAPHORE(sem)` (le compteur est initialisé à 1)
  - Dynamiquement avec la fonction `void sema_init(struct semaphore *sem, int initial_val)`

Opérations

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_killable(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
int down_timeout(struct semaphore *sem, long jiffies);

void up(struct semaphore *sem);
```

Plan

Introduction

Variables atomiques

Mutex

Spinlock

Sémaphores

Conclusion

Exemple d'utilisation : exclusion mutuelle

```
// Warning: no error management (empty or full stack)
struct stack_struct {
   int idx;
   int stack[64];
   spinlock_t lock;
};

void push(struct stack_struct *s, int v) {
   spin_lock(&s->lock);
   /* Beginning of critical section */
   s->stack[s->idx] = v;
   s->idx += 1;
   /* End of critical section */
   spin_unlock(&s->lock);
}

void pop(struct stack_struct *s, int *v) {
   spin_lock(&s->lock);
   /* Beginning of critical section */
   s->idx -= 1;
   *v = s->stack[s->idx];
   /* End of critical section */
   spin_unlock(&s->lock);
}

void init_foo(struct stack_struct *s) {
   s->idx = 0;
   spin_lock_init(&s->lock);
}
```

Verrous et attente

```
mutex_lock_interruptible(&mutex); // Beginning of critical section
// ...
while (!condition) {
  mutex_unlock(&mutex); // Release the mutex before sleep

  wait_event_interruptible(wq, condition); // Wait

  mutex_lock_interruptible(&mutex); // Relock the mutex
}
// ...
mutex_unlock(&mutex) // End of critical section
```

- Pourquoi tester de nouveau la condition après le deuxième `mutex_lock` ?

Verrous et attente

```
mutex_lock_interruptible(&mutex); // Beginning of critical section
// ...
while (!condition) {
  mutex_unlock(&mutex); // Release the mutex before sleep

  wait_event_interruptible(wq, condition); // Wait

  mutex_lock_interruptible(&mutex); // Relock the mutex
}
// ...
mutex_unlock(&mutex) // End of critical section
```

- L'évaluation de la condition lors du réveil est réalisé sans posséder le mutex
- De plus cette évaluation et le verrouillage du mutex ne sont pas réalisés de façon atomique
- Un autre thread pourrait dans l'intervalle altérer les structures de données de telle manière que la condition soit devenue fausse
- D'où la nécessité de tester de nouveau la condition une fois le mutex pris

Problèmes

- L'utilisation de *spinlocks*, de *mutexes* ou de *sémaphores* peut poser des problèmes d'interblocage (*deadlock*)
  - Si une fonction a verrouillé un mutex (ou un spinlock ou un sémaphore) puis appelle une autre fonction qui essaie elle aussi de verrouiller le même mutex, on se retrouve avec un blocage
  - Si une première fonction verrouille le mutex A (ou le spinlock ou le sémaphore) puis essaie de verrouiller le mutex B pendant qu'une seconde fonction a verrouillé le mutex B puis essaie de verrouiller le mutex A, les deux fonctions vont bloquer en essayant de prendre chacun l'autre mutex

Problèmes (suite)

- La solution au second problème est de toujours verrouiller les ressources dans le même ordre (si par exemple il y a deux mutexes A et B, il faut décider par exemple de toujours verrouiller A puis B si jamais on a besoin de verrouiller les deux)
- Ces problèmes ne sont pas spécifiques au noyau mais concernent toutes les applications multithreads
  - Cependant, contrairement à une application qui serait bloquée, si tout ou partie du noyau est bloqué, il ne reste guère de solutions autres que de redémarrer

Conclusion

- Pour faire de l'exclusion mutuelle
  - Sections critiques courtes : *Spinlock* (attente active)
  - Sections critiques plus longues : *Mutex* (mise en sommeil)
- Pour gérer de multiples ressources (modèle producteur/consommateur par exemple)
  - *Sémaphores* (mise en sommeil)