Synchronisations

Par Mathieu et Mathieu et le troisième Mathieu : Julien

Problématique du compte en banque

- Nous prenons pour exemple un programme qui permet d'ajouter 100€ sur un compte en banque.
  - le programme est situé sur les serveurs de la banque
  - plusieurs accés à ce code sont possibles : par exemple via une application android et via le site web de la banque

Problématique du compte en banque

```
lire (N, A);
N := N + 100;
écrire (N, A);
```

Avec deux processus différents

```
processus P1                          processus P2
lire (N, A);

                                      lire (N, A);
                                      N := N + 100;
                                      écrire (N, A);
N := N + 100;
écrire (N, A);
```

Section critique

- “Ensemble de suites d’instructions qui peuvent produire des résultats imprévisibles lorsqu’elles sont exécutées simultanément par des processus différents”

  => notion d’exclusion multiples

Test and Set

- Permet de protéger un espace de la mémoire en cas d'accès concurrents

- Si plusieurs processus tentent d'accéder à la même mémoire
  - -> Aucun autre processus ne peut commencer tant que le premier processus n’a pas terminé.

Test and Set example (1)

```
TestAndSet: // Assume it is called as TestAndSet(&lock).
            // This code is gcc/linux/intel x86 specific.

    // Preserve ebx, which is about to be modified.
    pushl    %ebx

    movl    8(%esp), %ebx       # &lock to ebx
    movl    $1, %eax            # 1 (true) to eax

    // Swap eax and lock. Value 1 (true) is copied to lock, eax receives
    // old lock value
    xchg    %eax, (%ebx)    # Atomically exchange eax with lock. The
                            # atomicity of xchg is what guarantees that
                            # at most one process or thread can be
                            # holding the lock at any point in time.

    // Restore ebx
    popl    %ebx

    // Return value (old value of lock) is already in register eax

    ret
```

Test and Set example (2)

```
volatile int lock = 0;

void Critical() {
    while (TestAndSet(&lock) == 1);
    critical section //only one process can be in this section at a time
    lock = 0 //release lock when finished with the critical section
}
```

SpinLock

le **spinlock** ou **verrou tournant** est un mécanisme simple de synchronisation basé sur l'attente active. Il est utilisé par exemple par les handlers d'interruption

Un spinlock est un mécanisme d'exclusion mutuelle qui ne peut avoir que deux valeurs locked et unlocked.

```
static DEFINE_SPINLOCK(xxx_lock);
        unsigned long flags;
        spin_lock_irqsave(&xxx_lock, flags);
        ... critical section here ..
        spin_unlock_irqrestore(&xxx_lock, flags)
```

Attention, le spinlock est un mécanisme qui peut être très gourmand en ressources

Verrous

- Verrouiller (v): acquiert le verrou v ou met en attente le processus

- Déverrouiller (v): déverrouille le verrou v que le processus possédait.

Ces opérations sont indivisibles

Sémaphore

- Sémaphore = distributeur de jetons
- Nombre de jetons : fixe et non renouvelable
- -> Les processus doivent restituer leur jeton après utilisation

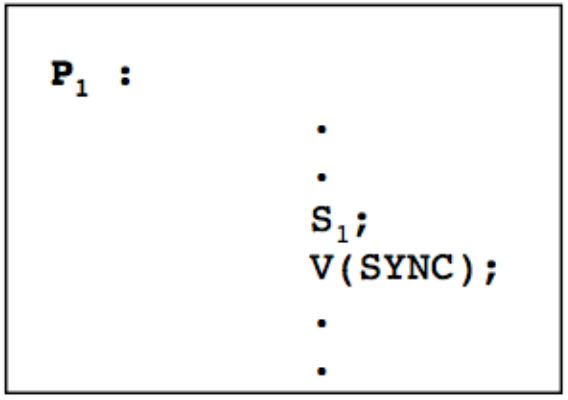
Un sémaphore contient un int constituant le nombre de jetons et qui sera décrémenté par chaque processus l'utilisant.

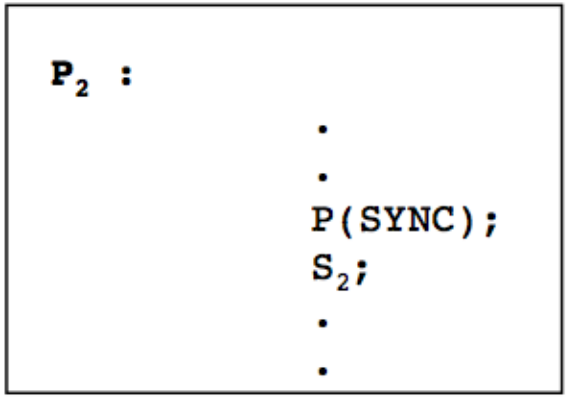
Un même sémaphore peut être utilisé dans plusieurs threads. Celui qui prend et jeton et celui qui le rend peuvent être différents

Opérations sur les sémaphores

- P : Prend un jeton s'il y en a ou bloque le processus.

- V : Rend un jeton.





Sémaphore

```
Semaphore::Semaphore(int _val){
    value = _val;
    c = PTHREAD_COND_INITIALIZER;
    m = PTHREAD_MUTEX_INITIALIZER;
}
```

```
int Semaphore::P(){
    if(value <= 0){
        pthread_cond_wait(&c, &m);
    }
    value--;
}
```

```
int Semaphore::V(){
    value++;
    if(value > 0){

pthread_cond_signal(&c);
    }
}
```

Synchronisation : wait() & notify()

- *wait()* : le thread qui possède l'accès attend une demande.

- *notify()* : le thread demande un accès. Si d' autres threads avaient appelé *wait()*, ils sont réveillés.

Exclusion mutuelle - Java

Permet de réserver l'accès à un objet à un seul Thread.

```
le code de methode1 et de methode2 est équivalent.

synchronized void methode1() {
    // section critique...
}

void methode2() {
    synchronized(this) {
        // section critique...
    }
}
```

Mutex

- Il y a au plus une entité en section critique
- Un mutex ne peut être utilisé que par un seul thread
  - celui qui prend le jeton doit être celui qui le rend
- Différentes implémentations :
  - sémaphore avec 1 jeton
    - Dans ce cas, il faut modifier le sémaphore satisfaire la contrainte d'unicité des threads
  - lock
  - spinlock

DeadLock