Systèmes d'exploitation - Synchronisations III

Othmane AJDOR

2018-2019

Table des matières

1 Grain

"Grain" calcul/cout de gestion ou de communications

2 Synchronisation dans les OS

2.1 Antiquité

2.1.1 Projet système

On disposait d'un seul processeur, soit une activité à la fois.
Il ne faut interrompre des activités qui touchent à des ressources pour lancer une deuxieme. Pour cela, on met en place des sections critiques (sémaphores) dans l'OS et masquer les interruptions (!= exception -déclenché par le processeur lui meme-) pendant l'appel de l'OS.
Les interruptions ne seront prises en charges qu'à la sortie du bout de code comme on les a masquées.Les exceptions se font une à la fois, on peut pas faire deux codes (user et section critique dans l'OS).

2.1.2 Linux 1.0

Gestion de multi-processeurs.
Il faut que le noyau fonctionne meme si les processeurs veulent executer des fonctions du systeme.
Pour ce faire, on a choisi de placer un BKL (bit kernel lock) à chaque entrée de l'OS. Ce lock est une attente active avec les fonctions atomiques.
Ce modèle ne passe pas à l'echelle à cause de l'attente active.

2.1.3 Aller plus vite

Pour aller plus vite, on parrallelise le noyau => remplacer le BKL (unique) par des (beaucoup) petits locks.
Pour sortir les tests de synchronisation du noyau => test en mode user ou blocage, liberation de threads en mode kernel.

Attente active :

```
int lock = false;
while (atomic_load(lock)); // /!\ BUG
    atomic_store(lock, 1); // /!\ BUG
SC
    atomic_store(lock, 0);
```

test_and_set() => tester une valeur, si elle vaut 0, je la met à 1 et je retourne l'ancienne valeur

```
while (test_and_set(lock));
    SC
```

CMP_XCHG(v, old, new) :

```
if v==old
    mettre v a new
renvoyer l ancienne valeur de v

/* Verifie si la valeur de old a été modifiée par nous meme ou dans une autre
partie du code */
atomic_inc(v){
    old = atomic_load(v);
    while(cmp_exchange(v, old, old+1) != old){
        old = atomic_load(v);
    }
}
```

Load Linked :

```
load_linked(v);
//...
//...
bool store_conditionnal(v, new); // bool reussi ou pas
```

2.1.4 Futex

Quand on fait un wait, on attend à ce que v ait la valeur attendue

```
futex_wait(&v, attendue); // attendu est la valeur liée au test atomic coté user
futex_wake(&v, nbWake); // Identique à signal où nbWake
// est nombre de threads à reveiller
```

```
int mutex=0;
mtx_lock(*m){
    while((c_atomic_fetch_add(&m, 1)) != 0){ // coute 10 cycles
        futex_wait(&m, c); // coute 20k cycles
    }
}

unlock(*m){
    atomic_store(&m, 0); // Ajoute barriere memoire qui valide les caches (10 cycles)
    futex_wake(&m, 1); // reveiller un thread (20k cycles
    )
}
```