Systèmes d'exploitation : moniteurs

William SCHMITT

2018-2019

1 Introduction

Les **moniteurs** permettent de faire de la synchronisation de **threads** : un même programme, dont la mémoire est partagée.

On considère un ensemble de fonctions en exclusion mutuelle. On peut faire attendre un thread, tant que l'état d'un système n'est pas propice à son exécution. Cette contradiction est gérée par des variables de condition, dont le rôle est d'organiser une file d'attente de threads.

2 Exercices

Il s'agit de comprendre le fonctionnement de ces cas, qui sont déjà implantés dans les bibliothèques standard.

2.1 Barrière à N

Des threads attendent à une barrière, lorsqu'il y en a N qui sont à la barrière, tous les N continuent.

On a besoin de :
— Mutex pour l'exclusion mutuelle
— Compteur pour connaître le thread
— Variable de condition pour savoir quand attendre/partir

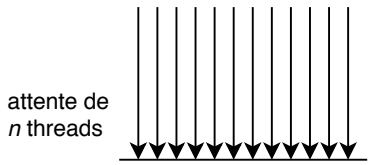

Figure 1 – Barrière à N

```
const N = 42;
int compteur = 0
mtx_t m;
cnd_t c;

void barriereN() {
    mtx_lock(&m);
    compteur++;
    while (compteur < N) { // On aurait également pu utiliser if ici
        // le compteur++ peut également être là
        cnd_wait(&c, &m);
    }
    cnd_signal(&c);
    mtx_unlock(&m);
}
```

Remarque : `cnd_signal()` réveille un thread à la fois. On observe donc ici un réveil en cascade (le dernier réveille le premier, qui réveille le second etc.).

Remarque 2 : le $N+1$ème thread passe directement, la barrière est donc à usage unique.

2.2 Allocateur

On ne s'occupe pas de la façon d'utiliser les fonctions, il y a cependant des détails à connaître :
— Les ressources ne sont pas infinies : il est possible de devoir attendre dans `allocation`
— Il n'y a pas d'attente pour `liberation`
On se contente de réveiller tous les threads en attente dans `allocation`, et on laisse les threads vérifier si assez de ressources sont disponibles.

Une autre fonction existe : `void cnd_broadcast(cnd_t * c)`, qui débloque toute une file.

```
    int resource = N;
    mtx_t m;
    cnd_t c;

    void allocation(int n) {
        mtx_lock(&m);
        while (resource < n) {
            cnd_wait(&c, &m);
        }
        resource -= n;
        mtx_unlock(&m);
```

```
}

void liberation(int n) {
    mtx_lock(&m);
    resource += n;
    cnd_broadcast(&c);
    mtx_unlock(&m);
}
```

Ce code est susceptible de créer des famines : pour un ensemble de threads demandant 1, sauf un qui demande 2, ce dernier pourrait ne jamais partir.

On peut essayer de faire du réveil en cascade. Dans la sémantique de Hoare le signal est bloquant, mais en réalité signal n'est jamais implanté de façon bloquante, ce code est donc **FAUX**. Dans le cas où l'on a 2 threads qui demandent trop de ressources, ils peuvent se renvoyer la balle constamment, se réveillant l'un l'autre.

```
// CODE FAUX - ILLUSTRATION
int resource = N;
mtx_t m;
cnd_t c;

void allocation(int n) {
    mtx_lock(&m);
    while (resource < n) {
        cnd_wait(&c, &m);
        cnd_signal(&c); // réveille le suivant
    }
    resource -= n;
    mtx_unlock(&m);
}

void liberation(int n) {
    mtx_lock(&m);
    resource += n;
    cnd_signal(&c); // réveille le premier
    mtx_unlock(&m);
}
```

2.3 Producteur-consommateur

Un producteur est un tampon (de taille bornée). On peut l'implanter de diverses manières, mais on utilise ici un tampon circulaire, un tableau de taille N utilisé avec deux indices : un indice de lecture et un indice d'écriture. On ne peut pas différencier le cas "tableau vide" du cas "les deux indices ont

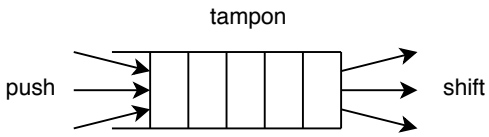

FIGURE 2 – Tampon

été incrémentés n fois". On se munit donc d'une autre variable : le nombre de messages déposés dans le tableau.

On sépare la file en deux, de façon à savoir qui on réveille.

```
int iecr = 0;
int ilect = 0;
int nb = 0;
mtx_t m;
cnd_t fp; // File pour les producteurs
cnd_t fconso; // File pour les consommateurs

void push(Msg msg) {
    mtx_lock(&m);
    mtx_unlock(&m);
}

Msg shift() {
    mtx_lock(&m);
    mtx_unlock(&m);
}
```