520 Outils de synchronisation

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Concurrence et sections critiques

On y arrive à la main, mais

- C'est compliqué
- C'est bas niveau
- C'est douteux d'un point de vue performance (attente active)

Peut-on faire mieux ?

- Le système d'exploitation est $\pm$ capable de se débrouiller
- Qu'en est-il des processus et threads ?

Solution : Un nouveau niveau d'indirection

- Système d'exploitation, bibliothèques et langages
- Ils fournissent des **outils**

$\rightarrow$ Services et des modèles de synchronisation clé en main

- Les développeurs peuvent les utiliser

Outils spécialisés

- La programmation concurrente **reste** complexe
- Ces outils ne suppriment pas les difficultés fondamentales
- Au mieux, ils les transforment et les déplacent...

Besoin de performance

- Rappel : les appels système coutent cher
- Une partie des mécanismes est faite en espace utilisateur
  $\rightarrow$ Langages et bibliothèques
- Une autre partie en mode noyau, par le système d'exploitation
  $\rightarrow$ ordonnancement et états d'exécutions des processus

Concrètement

- Implémentés avec les techniques primitives précédentes
- Garantissent l'efficacité et la fiabilité

$\rightarrow$ Bienvenue dans la programmation concurrente moderne !

$\rightarrow$ Rappel, ceci n'est pas un cours de programmation concurrente

Éliminer l'attente active

Le système d'exploitation gère le cycle de vie des threads

Solution à l'attente active

- Un thread veut entrer en section critique déjà occupée
  On le bloque (passage de l'état actif à bloqué)
  On appelle l'ordonnanceur
- Un thread sort d'une section critique
  Un autre thread était en attente ?
  On le réveille (passage à l'état prêt)
  Et on appelle l'ordonnanceur
- Le tout de façon performante !

Question

- Quels sont les cas où l'attente active est préférable à un changement de contexte ?

Mutex (ou verrou, *lock*), de *mutual exclusion*

- Concept général de verrouillage de section critique
- Mais détails spécifiques en fonction du contexte (système d'exploitation, bibliothèque, langage de programmation)

Opérations générales

- Verrouiller : ça entre ou ça attend
- Déverrouiller : ça débloque les autres
- Tenter : ça entre ou ça échoue

Variations

- Actif (*spinlock*) ou bloquante (passage à l'état bloqué)
- Rapide (un booléen), récursif (un compteur), avec détection d'erreur (on y reviendra)

Mutex pthread

- Fourni de base chez `pthreads(7)`
- `pthread_mutex_lock(3)`, `pthread_mutex_unlock(3)`, `pthread_mutex_trylock(3)`, etc.
- Limités aux threads d'un même processus
- RTFM pour les détails

```
#include<pthread.h>

long i; // Ressource partagée
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;

void inc(void) {
  pthread_mutex_lock(&mut); // on verouille
  i++; // on manipule
  pthread_mutex_unlock(&mut); // on déverouille
}
```

Sémaphore

- Concept historique spécifique (Dijkstra, 1962)
- C'est un compteur de ressources
- Le compteur bloque s'il n'y a plus de ressource
- Sémantique atomique garantie (détail d'implémentation)
- Pas d'attente active et pas de famine
- Cas particulier: sémaphore binaire (deux valeurs possibles), ressemble au mutex d'un point de vue de l'implémentation

Sémaphore : pseudo-code

```
Sémaphore { entier val; liste_processus L; };
```

Demander(Sémaphore S)

S.val -= 1;
si *S.val* $<$ *0* **alors**
    ajouter demandeur à S.L;
    le passer à bloqué;
fin

Libérer(Sémaphore S)

S.val += 1;
si *S.val* $<=$ *0* **alors**
    enlever un processus de S.L;
    le passer à prêt;
fin

Remarque de vocabulaire

- Demander = Entrer = Down = P = *proberen* = tester
- Libérer = Sortir = Up = V = *verhogen* = incrémenter

Sémaphores Unix

- Ressources gérées globalement au niveau du système
- Persistant jusqu'à l'arrêt du système ou une libération explicite
- Partageables entre threads, processus et utilisateurs
- Sémaphores POSIX : `sem_overview(7)`
- Sémaphores System V : `semget(2)`, `semop(2)`

```
#include <semaphore.h>
long i; // Ressource partagée
static sem_t sem;
void inc_init(void) { // initialisation
  sem_init(&sem, 0, 1); // initialise à 1
}
void inc(void) {
  sem_wait(&sem); // verrouillage
  i++;            // on manipule
  sem_post(&sem); // dévérouillage
}
```

Sémaphore vs Mutex

Sémaphore

- Compteur de ressources : atomique, efficace et équitable
- Ceux qui incrémentent sont pas forcément ceux qui décrémentent

Mutex système

- Délimite une section critique qui protège une ressource partagée
- Le thread qui déverrouille est celui qui a fait le verrouillage initial
- Information utile pour le système d'exploitation

Avantages des mutex système

- Déverrouillage des mutex d'un thread qui termine
- Inversion de priorité possible (on y reviendra)
- Vérification d'erreur possible:
  - Un thread déverrouille un mutex sans l'avoir verrouillé
  - Situation d'interblocage (on y reviendra)

Autres outils et techniques de synchronisation

- Variable de condition (file d'attente + service de réveil)
- Moniteur (sous-programmes + mutex implicite + variables de condition)
- Barrière
- Verrou lecture-écriture
- RCU (*read-copy-update*) : technique sans verrouillage
- Structures de données parallèles clé en main
- Opérations atomiques (C11)
- Etc.

Ce sont des outils et abstractions de programmation

Pour plus de détails

- Programmation concurrente et parallèle (INF5171)
- Programmation parallèle haute performance (INF7235)
- *Is Parallel Programming Hard, And, If So, What Can You Do About It?*, Paul E. McKenney.

Futex (*fast userspace mutex*)

- Bloque un processus jusqu'à un réveil explicite
- Bas niveau et délicat

→ Erreur classique : on bloque un processus
Pile au moment où la condition du blocage disparait

- Sert aux bibliothèques pour implémenter les autres mécanismes
  → Mutex pthread et autre
- `futex(2)` sous Linux

Exemple tentative futex

```
#define _GNU_SOURCE
#include <sys/syscall.h>
#include <unistd.h>
#include <linux/futex.h>
#include "myatomic.h"
long i; // ressource partagée
int flag; // 0=libre; 1=occupé
void inc(void) {
  while(xchg(&flag, 1) == 1) {
    // on s'endort si occupé
    syscall(SYS_futex, &flag, FUTEX_WAIT, 1, 0, 0, 0);
  }

  i++; // on manipule

  xchg(&flag, 0);
  // on réveille un endormi, s'il y en a
  syscall(SYS_futex, &flag, FUTEX_WAKE, 1, 0, 0, 0);
}
```

Exemple futex mieux

```
#define _GNU_SOURCE
#include <sys/syscall.h>
#include <unistd.h>
#include <linux/futex.h>
#include "myatomic.h"
long i; // ressource partagée
int flag; // 0=libre; 1=occupé; 2=endormi
void inc(void) {
  int c = cmpxchg(&flag, 0, 1);
  if(c != 0) {
    if (c != 2) c = xchg(&flag, 2);
    while (c != 0) {
      syscall(SYS_futex, &flag, FUTEX_WAIT, 2, 0, 0, 0);
      c = xchg(&flag, 2);
    }
  }
  i++; // on manipule
  if(xchg(&flag, 0) == 2)
    syscall(SYS_futex, &flag, FUTEX_WAKE, 1, 0, 0, 0);
}
```