Architecture et Systèmes

Stefan Schwoon

Cours L3, 2021/2022, ENS Paris-Saclay

Programmation concurrente

Motivation :

- séparation des tâches
- meilleure efficacité (si le CPU dispose de plusieurs cœurs)

Technique connue : les processus

- sécure, puissant, mais lourd
- communication entre processus par pipe/signaux

Alternative : les *thread* (ou *processus légers*)

Les threads

Un thread est un fil d'exécution *à l'intérieur d'un processus*.
Normalement, un processus ne possède qu'un seul thread.

Rappel: Les processus sont des unités d'exécution avec leurs propres ressources (mémoire, signaux, . . .), entièrement indépendants l'un de l'autre.

Par contre, les threads d'un processus se *partagent* la plupart de leurs ressources.

Ressources privés et partagés

Ressources propres d'un thread :

son propre compteur de programme, son jeux de registres et sa propre pile

ses variables locales (non `static`), allouées sur la pile

code de retour (un pointeur `void*`)

Ressources partagés :

tout le reste de la mémoire (code et tas)

leurs fichiers ouverts

les signaux

Travailler avec les threads

Librairie standard : Posix threads (Pthreads)

Pour compiler, inclure `pthreads.h` et utiliser `gcc -pthread`.

Voir `pthreads(7)` pour un survol de la thématique.

Remarques

Il n'existe aucune hiérarchie entre les threads d'un même processus !

Ordonnancement :

Kernel threads: Les threads sont connus au noyau qui leur accorde les créneaux de calcul.

User threads: (disponible dans certains version de Linux)
Le noyau ne contient que des processus ;
les threads réalisés par un ordonnanceur à l'intérieur du processus (donc à l'extérieur du noyau, en mode utilisateur).

Un thread possède un identifiant (type `pthread_t`).
Cet identifiant peut être utilisé avec les fonctions de la famille pthread.

Création d'un thread

N'importe quel thread peut en créer un autre avec `pthread_create`.

Le nouveau thread exécute une fonction de départ fournie comme paramètre.

Cette fonction de départ prend un argument de type `void*` et renvoie un `void*`.

Terminaison d'un thread

Un thread termine . . .

à la fin de sa fonction de départ;

avec un appel `phread_exit`;

si un autre thread appelle `pthread_cancel`.

Code de sortie:

Sur termination, le noyau enregistre un code de sortie.

Comme pour les processus, ce code peut être récupéré par un autre thread (`pthread_join`).

`pthread_detach` rend un thread "non-joignable" :
pas de code de sortie, ni de zombie.

Vie et mort d'un processus avec threads

Un processus est vivant lorsqu'il possède au moins un thread vivant.

Les actions suivantes tuent un processus avec tous ses threads :

appel d'`exit` (par n'importe quel thread);

le thread principal termine la fonction `main`
(appelle `exit` implicit) ;

le processus reçoit un signal terminant.

Threads vs processus

Avantages des threads : plus efficace

Communication entre threads plus faciles (par mémoire partagée au lieu des fichiers/tubes/signaux).

Création des threads moins couteux pour le noyau.

Inconvenients : moins sécure

Une erreur dans un seul thread peut tuer le processus entier.

Certains appels système ne sont pas thread-safe, deux threads ne peuvent pas s'en servir simultanément (voir *pthreads(7)* pour une liste).

Attention aux daccès mémoire (*situations de compétition*) !

Problèmes de programmation concurrente

Compétition pour les réssoures (accès/modif de données, périphériques)
$\rightarrow$ problème d'exclusion mutuelle

Coordination: établir un processus distinct parmi des pairs
$\rightarrow$ élection d'un leader

Signaler la présence / absence de données:
$\rightarrow$ problème de producteur / consommateur

Programmation concurrente

Les solutions à ces problèmes dépendent du contexte :

- Concurrence entrelacée ou vraie (calcul multi-cœur ou distribué)
- Mémoire partagée ou pas
- Accès mémoire en écriture/lecture seulement
- Moyens de communication (synchrone/asynchrone/délai borné)
- Existence d'une autorité centrale qui peut résoudre des conflits (p.ex. noyau, serveur).

Exclusion mutuelle

Modèle abstract :

On a un ensemble de processus/threads qui possède tous des sections critiques (une partie du code).

On doit assurer qu'au plus un seul processus est dans une section critique en même temps.

On souhaite une structure de données (on l'appèle mutex) qui permet au moins les opérations suivantes:

acquérir : si plusieurs processus essayent à obtenir le mutex, un seul réussira.

relàcher : un processus qui detient le mutex le rend accessible aux autres.

Propriétés intéressantes : correction, absence de blocages, justice

Propriétés d'un mutex

Correction: Un seul processus/thread peut detenir le mutex en même temps.

Absence de blocages: Si plusieurs processus/threads tentent d'obtenir le mutex, au moins un réussira.

Justice: Si un processus tente d'obtenir un mutex, il réussira à un moment donné.

Algorithme de Peterson

Solution pour deux processus avec mémoire partagé (trois bits)

On suppose que la lecture/écriture d'un bit est atomique.

Variables:

- `flag[0]`: premier processus veut entrer dans une section critique
- `flag[1]`: deuxième processus veut entrer dans une section critique
- `victim`: pour résoudre des conflits

Algorithme de Peterson

Au départ : `flag[0] = flag[1] = 0;`

Code du processus i=0,1 (autour de la section critique) :

```
autre = 1-i;
flag[i] = 1;
victim = i;
while (victim == i && flag[autre]);
...  critical section ...
flag[i] = 0;
```

Remarque: La conjonction (`&&`) peut être non-atomique et évaluée dans n'importe quel ordre.

En supposant que les processus terminent toujours leurs section critiques, l'algorithme de Peterson est ...

correct (un seul processus peut être critique à la fois) ;

juste (tout processus réussit finalement à entrer dans sa section critique);

libre de blocages.

Il est possible de généraliser le principe à n participants qui font $n - 1$ tours d'élimination.

Problèmes

Un algorithme tel que Peterson résout le problème en absence d'autorité centrale, mais présente des inconvenients:

- Du code compliqué à écrire autour de chaque accès.
- Nécessite la mémoire partagée.
- Assez lourd pour plusieurs processus.
- Facile de se tromper.
- Attention aux réordonnancements dans le processeur !

Avec le noyau comme autorité centrale, POSIX propose plusieurs solutions (sémaphores, spinlocks).

Sémaphores

Un sémaphore est une structure de donnée gérée par le noyau qui offre une solution si tous les processus sont dans un même ordinateur.

Gère un compteur de *créneaux* disponibles, avec les opérations suivantes :

Init(n), où n est un nombre de *créneaux* initiaux

Wait: si un créneaux est disponible, l'obtenir; sinon on attend

Post: libérer un créneaux

Implémentation d'une sémaphore

Naïvement :

```
Init(n) { ctr = n; }

Wait() { while (ctr == 0); ctr = ctr-1; }

Post() { ctr = ctr+1; }
```

Notes :

Les opérations sont "atomiques" (le noyau utilise un mécanisme de mutex pour l'assurer).

pendant l'attente, le thread/processus concerné est mis en sommeil (pas d'attente active).

Cas d'usage typique pour sections critiques

```
Init(1);

while (1) {
    ...;
    Wait();
    Critical1();
    Post();
    ...;
}
```

```
while (1) {
    ...;
    Wait();
    Critical2();
    Post();
    ...;
}
```

Mettre Wait et Post autour des accès.

Sémaphores dans Unix

Supporté par le noyau, voir `sem_overview` (7):

Sémaphores anonymes (entre threads/processus pére et fils):
`sem_init`, `sem_wait`, `sem_post`

Sémaphores nommés (dans tout le système):
`sem_open`, `sem_unlink`

Spinlocks

Alternative aux sémaphores, avec attente active.

peut être efficace dans un contexte de vraie concurrence quand les attentes sont courtes.

ne jamais utiliser dans un contexte de concurrence entrelacée !

Voir : `pthread_spin_lock`