

# Architecture et Systèmes

Stefan Schwoon

Cours L3, 2020/2021, ENS Cachan

# Programmation concurrente

---

## Motivation :

séparation des tâches

meilleure efficacité (si le CPU dispose de plusieurs cœurs)

## Technique connue : les processus

sécuré, 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 jeu 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

(partiellement) les signaux

# Travailler avec les threads

---

Librairie standard : Posix threads (Pthreads)

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

Voir `pthread(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 certaines versions 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 `pthread_exit`;

par `pthread_cancel` depuis un autre thread.

## Code de sortie:

Dans les deux premier cas, 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.

# Avantages et inconvénients

---

Avantages des threads par rapport aux processus :

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

Création des threads moins coûteux pour le noyau.

Bref, c'est plus efficace.

Inconvénients :

Moins sécurisée - 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 *pthread(7)* pour une liste).

Attention lors des accès mémoire (*situations de compétition*) !

# Programmation concurrente

---

Les threads sont un exemple de *programmation concurrente*.

Autres exemples :

Arbitration dans le noyau

Protocoles de réseau

Dans toutes ces situations on se trouve devant les problèmes suivants :

Coordonner l'accès aux ressources partagés (exclusion mutuelle); éviter les situations de compétition, communication avec périphériques

Signaler qu'une ressource est prête à consommer

# 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).

# Premier exemple: Section critique

---

Modèle abstrait :

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'appelle **mutex**) qui permet au moins les opérations suivantes:

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

**relâcher** : 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 partagée (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.

# Extension à $n$ participants

---

Idée: exécuter l'algorithme de Peterson à  $n - 1$  niveaux où chaque fois filtre un participant.

Variables:

`level[0], ..., level[n-1]`: niveau du  $i$ -ème participant

`victim[1], ..., victim[n-1]`: la "victime" au niveau  $i$

Note : toujours pour mémoire partagée, mais avec lecture/écriture d'un entier atomique

## Extension: pseudocode

---

Initialement, `level` est 0 pour tous les niveaux.

Vue de participant numéro  $i=0,\dots,n-1$ :

```
for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while (( $\exists$  k != i: level[k] >= L) && victim[L] == i);
}
... critical section ...
level[i] = 0;
```

# Problèmes

---

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

Du code compliqué à écrire autour de chaque accès.

Il est facile de se tromper dans la programmation.

Nécessite la mémoire partagée.

Assez lourd pour plusieurs processus.

**Attention aux réordonnements 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) {                      while (1) {  
    ...;  
    Wait();  
    Critical1();  
    Post();  
    ...;  
}  
  
                                }  
                                ...;  
                                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 père 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`