

# 510 Section critique

## INF3173

### Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

# Problème de concurrence

- Les threads (et processus)
  - Le système d'exploitation lui-même
  - Sont fortement concurrents, voire parallèles
- Comment gérer cette concurrence **correctement** de façon programmaticative ?

## Objectifs

- Contrôler les situations de compétition
- Prévenir la corruption de ressources partagées
- Indépendamment du type de ressources
- Rester efficace



## Section critique = Zone de code

- Zone de code = morceau de programme
- Attention, pas forcément contiguë

## Section critique = Zone d'exclusivité

- Exécuté que par un seul thread\* maximum à la fois
  - Qui manipule une **ressource** potentiellement **partagée**
- On **protège** une ressource  
en **contraignant** l'exécution du code  
qui **manipule** cette ressource

---

\*Sans perte de généralité, on utilise « threads », mais ça s'applique pareil aux processus monothread, tâches noyau ou toute autre entité logicielle en cours d'exécution.

# Un thread en section critique

- N'est pas nécessairement actif à 100%
- Il peut faire des appels système bloquants (et devenir bloqué)
- Il peut être préempté (et devenir prêt)

## Section critique en bref

- Tant qu'un thread n'est pas sorti
- Aucun autre thread ne peut y rentrer

# Les 4 règles des sections critiques



- ① Au **maximum**, un **seul** thread à la fois en section critique
- ② Pas de **supposition** sur la vitesse ou le nombre de threads
- ③ Un thread **hors** section critique **ne bloque pas** les autres
- ④ Pas d'attente **infinie** pour entrer en section critique (**famine**)

# Solution qui marche pas

```
long i; // ressource partagée
int flag; // booléen protégeant la ressource

void inc(void) {
    while (flag) { } // on attend l'absence du flag
    flag = 1;        // on verrouille
    i++;            // on manipule
    flag = 0;        // on déverrouille
}
```

- Toute « stratégie » de ce type est à bannir !

## Question

- Lesquelles des 4 règles sont violées ?
- Trouvez un scénario (ordonnancement) où ça ne fonctionne pas

# Exclusion mutuelle stricte à tours

```
long i; // ressource partagée

extern int nb; // nombre de threads
int tour;      // à qui c'est le tour?

void inc(int t) { // on est le thread `t` (de 0 à nb-1)
    while (t != tour) {} // on attend notre tour
    i++;                  // on manipule
    tour = (tour+1) % nb; // on passe au suivant
}
```

## Question

- Lesquelles des 4 règles sont violées ?

# Solution de Peterson (1981)

- Pour deux threads seulement

```
long i; // ressource partagée

int flag[2]; // qui est intéressé ?
int tour;    // à qui le tour (si les deux en veulent) ?

void inc(int k) { // k c'est moi, !k c'est l'autre
    flag[k] = 1; // on veut entrer
    tour = !k;  // on est poli
    while (flag[!k] && tour == !k) { } // attente
    i++;        // on manipule
    flag[k] = 0; // on n'en veut plus
}
```

- Se généralise à un nombre quelconque de threads.





Instruction (ou indication) destinée:

- Aux processeurs  
Force les écritures et lectures du bon côté de la barrière
  - Au compilateurs C  
Prévient les optimisations qui changent la sémantique
  - Coût non négligeable
- Les détails dans d'autres cours...

## Exemples

- Instruction `mfence` en x86
- Pas de mot clé C standard
- `atomic_thread_fence` de C11 `stdatomic.h`
- Extension C de gcc : `__atomic_thread_fence`  
(et le plus ancien `__sync_synchronize`)



- Mot clé C `volatile`
- Déclare une donnée comme mutable par quelque chose d'autre (comme un autre thread)

## Pour le compilateur seulement

- Informe qu'une modification indépendante est possible
- Et qu'il doit éviter des optimisations
- Par défaut, le compilateur n'est pas conservateur !
- Les options de compilation changent le comportement du code



Quand le compilateur voit

« `int x=0; while(x==0) {}` »

- Si `x` n'est pas volatile  
On peut juste implémenter une boucle infinie
- Si `x` est volatile  
On **doit** tester `x` à chaque tour, « au cas où... »

## Bonne ou mauvaise chose ?

- La présence de `volatile` dans du code est souvent **douteuse**
- Son utilisation ne permet pas **magiquement** de résoudre les problèmes de concurrence
- Son **coût** est non négligeable
- *volatile considered harmful*

# Peterson + volatile + barrières mémoire

```
#include <stdatomic.h>

long i; // ressource partagée

volatile int flag[2]; // qui est intéressé ?
volatile int tour;    // à qui le tour ?

void inc(int k) { // k c'est moi, !k c'est l'autre
    flag[k] = 1; // on veut entrer
    tour = !k;  // on est poli
    atomic_thread_fence(memory_order_seq_cst); //barrière
    while (flag[!k] && tour == !k) { } // attente
    atomic_thread_fence(memory_order_seq_cst); //rebarrière
    i++; // on manipule
    atomic_thread_fence(memory_order_seq_cst); //rerebarrière
    flag[k] = 0; // on n'en veut plus
}
```

# Le matériel à la rescousse

## Idée : empêcher le changement de contexte

- Masquer les interruptions matérielles au niveau du processeur (dont l'horloge)
- Verrouiller le bus (en multiprocesseurs)

## Problèmes

- Grain grossier
- Couteux
- Seul le noyau peut faire ça  
(ok pour lui, mais pas pour les processus)

# Instructions machine atomiques

- C11 `_Atomic` et `stdatomic.h`, extension gcc ou assembleur
- Permet des manipulations **atomiques** :  
Indivisible pour l'observateur

## Exemple : incrément atomique

- `lock add (x86)`
- `__atomic_add_fetch (gcc)`
- `++` sur un type `_Atomic (C11)`

## Mise en œuvre matérielle



- Accès mémoire exclusif de la donnée le temps de l'exécution
- Exemple : verrouillage des lignes de cache mémoire
- Les détails dans un autre cours...

## Limites

- Coût non nul
- Seulement certaines instructions et valeurs simples

# Incrément atomique

## Extension gcc

```
long i; // Ressource partagée
void inc(void) {
    // On manipule sans verrous
    // Directement fonction built-in gcc
    __atomic_add_fetch(&i, 1, __ATOMIC_RELAXED);
}
```

## C11 avec `_Atomic`

```
_Atomic long i; // Ressource partagée
void inc(void) {
    i++; // on manipule
}
```

## En vrai ?

Le compilateur compile vers des instructions machine atomiques  
Il peut ajouter aussi des barrières

# Sections critiques plus grosses?

On implémente un verrou atomique

- Instructions « *test and set* », « *compare and exchange* »...
- x86: `xchg`, `lock cmpxchg`...
- gcc: `__atomic_test_and_set`, `__atomic_compare_exchange`...
- C11: `atomic_flag_test_and_set`, `atomic_compare_exchange`...

```
#include <stdatomic.h>
```

```
long i; // ressource partagée  
atomic_flag flag; // booléen protégeant la ressource
```

```
void inc(void) {  
    while(atomic_flag_test_and_set(&flag)) {}  
    i++; // on manipule  
    atomic_flag_clear(&flag);  
}
```

- C'est une version fonctionnelle de la « solution qui marche pas »



# Attente active (*spinlock*)



```
while (...) { }
```

- Quand ça fonctionne, ça reste inefficace
- Ça gaspille du temps processeur à **activement rien faire**

## Questions

Voici deux autres propositions :


- `while (...) { sched_yield(); }†`
- `while (...) { sleep(1); }`
- Pourquoi c'est pas vraiment beaucoup mieux ?
- Y a-t-il des cas où c'est même pire que la proposition initiale ?

---

<sup>†</sup>En gros `sched_yield(2)` force un appel à l'ordonnanceur pour éventuellement donner le processeur à un autre thread.

# Limites des propositions à date

## Objectifs

- Contrôler les situations de compétition ✓
- Prévenir la corruption de ressources partagées ✓
- Indépendamment du type de ressources ✓
- Rester efficace 

## Limites

- Approches purement algorithmiques limitées
- Instructions machine spécifiques peu portables
- Bricolage bas niveau
- Potentiellement inefficace (*spinlock*)

## Solution : Un nouveau niveau d'indirection

- Langages, bibliothèques et systèmes d'exploitation à la rescousse
- Ils fournissent des services et des modèles de synchronisation
- Que les développeurs peuvent utiliser