SYSTÈMES D'EXPLOITATION

SYNCHRONISATION DES PROCESSUS ET DES THREADS

🎓 3A - Cursus Ingénieurs - Dominante Informatique et Numérique
🏛 CentraleSupelec - Université Paris-Saclay - 2024/2025

Idir AIT SADOUNE
idir.aitsadoune@centralesupelec.fr

PLAN

Retour au plan - Retour à l'accueil

INTER PROCESS COMMUNICATION - IPC

- L'**OS** garantie l'**indépendance des processus**
  - par l'ordonnanceur CPU
  - par la gestion mémoire que l'on verra plus tard
- Un processus peut **communiquer** avec un autre processus ou avec des périphériques (fichiers, imprimantes, réseaux, ...).
- Il est alors nécessaire de mettre en oeuvre un mécanisme de **communication inter-processus**.

PRINCIPALES MÉTHODES DE COMMUNICATIONS

Méthode	Description
Signal	Un message système est envoyé d'un processus à un autre.
Pipe	Un canal unidirectionnel ; les données émises sont accumulées dans une mémoire tampon (FIFO).
File	Lecture/Écriture dans un fichier.
Socket	Un flux de données envoyé à travers une interface réseau à un autre processus.
Mémoire Partagée	Espace de mémoire alloué à plusieurs processus.
Moniteur/ Sémaphore	Une structure de synchronisation pour les processus travaillant sur des ressources partagées.

PROBLÈME DE LA CONCURRENCE

EXEMPLE

Soit la gestion d'un compte bancaire

- Une variable partagée `balance`
- Une fonction `add(1)` (`balance = balance + 1`)
- Une fonction `sub(1)` (`balance = balance - 1`)
- Le montant initial du compte est de 9€

```
1 ;add(1) or balance = balance + 1
2 mov eax, balance
3 add eax, 1
4 mov balance, eax
```

```
1 ;sub(1) or balance = balance - 1
2 mov eax, balance
3 sub eax, 1
4 mov balance, eax
```

PROBLÈME DE LA CONCURRENCE

EXEMPLE

```
1 ;add(1) or balance = balance + 1
2 mov eax, balance
3 add eax, 1
4 mov balance, eax
```

```
1 ;sub(1) or balance = balance - 1
2 mov eax, balance
3 sub eax, 1
4 mov balance, eax
```

On lance 2 threads en parallèle

- La première thread exécute `10 000 000` fois `add(1)`
- La deuxième thread soustrait `10 000 000` fois `sub(1)`

➠ **Résultat attendu** → `balance = 9€`

✘ **Résultat obtenu** → `balance = -98599€`

PROBLÈME DE LA CONCURRENCE

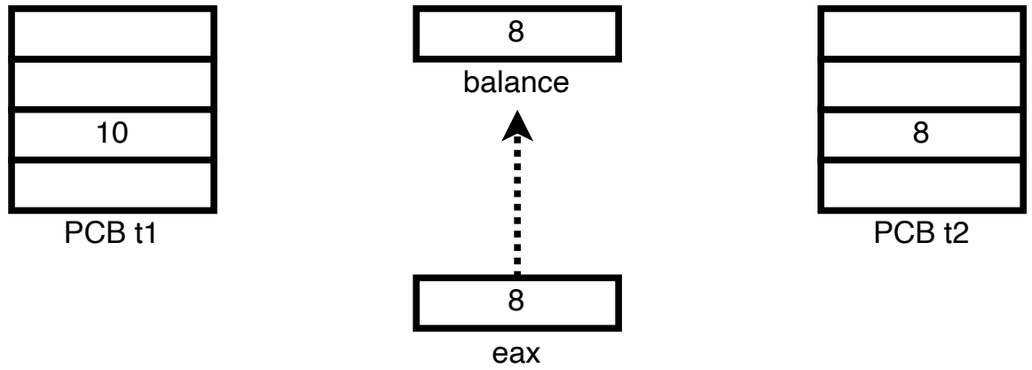
```
1 ;add(1) or balance = balance + 1
2 mov eax, balance
3 add eax, 1
4 mov balance, eax
```

👉

```
1 ;sub(1) or balance = balance - 1
2 mov eax, balance
3 sub eax, 1
4 mov balance, eax
```

Comment expliquer ces erreurs de calcul ?

- ➠ les entrelacements se font au niveau du code binaire
- ➠ couper **entre chaque instruction assembleur**



PROBLÈME DE LA CONCURRENCE

CONCLUSION

```
1 ;add(1) or balance = balance + 1
2 mov eax, balance
3 add eax, 1
4 mov balance, eax
```

```
1 ;sub(1) or balance = balance - 1
2 mov eax, balance
3 sub eax, 1
4 mov balance, eax
```

Après une itération $\longrightarrow$ `balance = 8`

alors qu'on s'attend à `balance = 9`

PROBLÈME DE LA CONCURRENCE

CONCLUSION

Situation de compétition

➠ **erreur** dépendant de l'enchaînement temporel d'événements impliquant une ressource partagée

➠ non déterministe
✘ difficile à détecter (**tests**)
✘ difficile à corriger (**debug**)

PLAN

Retour au plan - Retour à l'accueil

SECTION CRITIQUE

Lorsqu'il y a des **variables partagées**, il existe des **portions de code** qu'on ne veut pas pouvoir interrompre.

- des **zones du code** qui manipulent des **ressources partagées**
- ces zones sont appelées **sections critiques**

EXCLUSION MUTUELLE

- Lorsqu'on déclare une **section critique**, on doit garantir qu'**au plus un seul** processus/thread est dans la section critique.

  ➠ besoin de gérer l'exclusion mutuelle des **sections critiques**.
  ➠ **une seule section critique** peut être exécutée à la fois.

- Pour résoudre ce problème il faut un **système de verrou**.

LES PROPRIÉTÉS À RESPECTER

1. **Exclusion mutuelle** $\longrightarrow$ si une thread effectue sa section critique, alors **aucune autre thread** ne peut entrer en section critique.

2. **Déroulement** $\longrightarrow$ une thread, qui souhaite entrer en section critique, **ne peut pas décider** qui doit rentrer en section critique.

3. **Vivacité** $\longrightarrow$ une thread, qui souhaite entrer en section critique, **y rentre en temps borné**.

Il faut définir des **mécanismes** qui garantissent ces trois propriétés

PLAN

Retour au plan - Retour à l'accueil

UNE SOLUTION POUR 2 THREADS

PRINCIPES

Un contrôleur central qui:

- met en attente les processus/threads demandant l'accès en **SC**
- autorise l'accès en **SC** des processus/threads chacune à son tour

- respecte **les 3 objectifs de la synchronisation**

UNE SOLUTION POUR 2 THREADS

PRINCIPES

```
interface Mutex {
    abstract void commencerSC(int id);
    abstract void finirSC(int id);
    // id dans {0,1} num de la thread
}
```

```
//... code non critique
my_mutex.commencerSC(my_id);
//... code critique
my_mutex.finirSC(my_id);
//... code non critique
```

- Lorsque t_i invoque `commencerSC(i)`:
  - Vérifier que t_{1-i} n'est pas en SC (sinon, attendre)
  - Noter que t_i est en SC
- Lorsque t_i invoque `finirSC(i)`:
  - Noter que t_i n'est plus en SC

UNE SOLUTION POUR 2 THREADS

UNE PREMIÈRE IMPLÉMENTATION

```
1  class Mutex1 implements Mutex{
2      boolean[] est_SC = {false,false}; //pour noter qui est en SC
3
4      void commencerSC(int id){
5          while (est_SC[1-id])
6              ; //attendre ...
7          est_SC[id]=true;
8      }
9
10     void finirSC(int id){
11         est_SC[id]=false;
12     }
13 }
```

UNE SOLUTION POUR 2 THREADS

UNE PREMIÈRE IMPLÉMENTATION

```
1 void commencerSC(int id){
2     while (est_SC[1-id])
3         ; //attendre ...
4     est_SC[id]=true;
5 }
```

✘ Les deux threads sont en SC!
✘ Ce code contient lui-même des SC!

UNE SOLUTION POUR 2 THREADS

UNE DEUXIÈME IMPLÉMENTATION

```
1  //Mettre en attente la thread demandeuse si l'autre est déjà en SC
2  class Mutex2 implements Mutex{
3      boolean[] est_SC = {false,false};
4
5      void commencerSC(int id){
6          est_SC[id]=true; //on commence par noter la SC !
7          while (est_SC[1-id])
8              ;
9      }
10
11     void finirSC(int id){
12         est_SC[id]=false;
13     }
14 }
```

UNE SOLUTION POUR 2 THREADS

UNE DEUXIÈME IMPLÉMENTATION

```
1 void commencerSC(int id){
2     est_SC[id]=true;
3     while (est_SC[1-id])
4         ;
5 }
```

✘ **Problème** → interblocage

UNE SOLUTION POUR 2 THREADS

UNE TROISIÈME IMPLÉMENTATION

```
1  //Ajout de tours de priorité
2  class Mutex3 implements Mutex{
3      boolean[] est_SC = {false,false};
4      int tour = 0;
5
6      void commencerSC(int id){
7          tour=1-id;
8          est_SC[id]=true;
9          while (est_SC[1-id] && tour==1-id)
10             ;
11     }
12
13     void finirSC(int id){
14         est_SC[id]=false;
15     }
16 }
```

UNE SOLUTION POUR 2 THREADS

UNE TROISIÈME IMPLÉMENTATION

```
1 void commencerSC(int id){
2     tour=1-id;
3     est_SC[id]=true;
4     while (est_SC[1-id] && tour==1-id)
5         ;
6 }
```

✔ Une seule thread est passée, l'autre est en attente

ATTENTE ACTIVE

```
while (est_SC[1-id] && tour==1-id)
    ;
```

✘ Les processus font de l'**attente active**.
⟶ utilisation inutile du processeur

✔ Des **mécanismes de haut niveau** sont offerts par les **OS**

MÉCANISMES DE HAUT NIVEAU

Méthode	Description
Signal	Un message système est envoyé d'un processus à un autre.
Pipe	Un canal unidirectionnel ; les données émises sont accumulées dans une mémoire tampon (FIFO).
File	Lecture/Écriture dans un fichier.
Socket	Un flux de données envoyé à travers une interface réseau à un autre processus.
Mémoire Partagée	Espace de mémoire alloué à plusieurs processus.
Moniteur/ Sémaphore	Une structure de synchronisation pour les processus travaillant sur des ressources partagées.

PLAN

Retour au plan - Retour à l'accueil

MONITEUR

Un moniteur est un module constitué de:

- objets inaccessibles de l'extérieur
- fonctions manipulant l'objet en exclusion mutuelle

MONITEURS EN JAVA

- Dans la JVM de **Java**, on peut déclarer que des blocs de code sont en **exclusion mutuelle**.
  - utilisation du mot clé `synchronized` $\longrightarrow$ **un verrou** (**lock**) sur un objet
  - une seule thread dans les blocs `synchronized` sur le même objet
- Une thread, qui possède un verrou, peut exécuter n'importe quelle méthode, vérouillée ou non (**verrou récursif**).
- Une thread peut **verrouiller plusieurs objets**, ceci peut provoquer une situation **d'interblocage**.
- Tout bloc **non** `synchronized` (**non vérouillé**) peut être appelé par n'importe qui n'importe quand.

MONITEURS EN JAVA

EXEMPLE

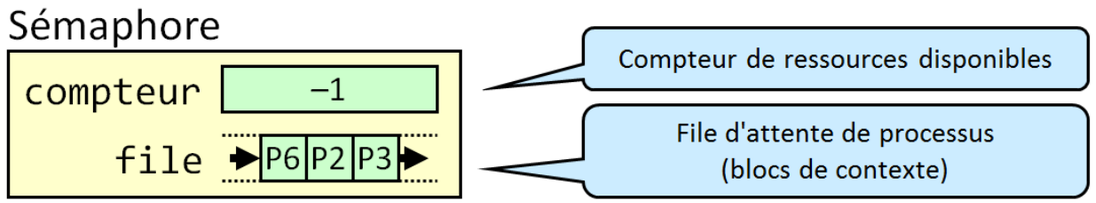
```java
public class Account {
    private int value;

    public Account(int i) {
        this.value = i;
    }

    synchronized public void add(int v){
        this.value = this.value + v;
    }

    synchronized public void sub(int v){
        this.value = this.value - v;
    }
}
```

SÉMAPHORE - DIJKSTRA - 1962

- **Un sémaphore** définit un objet partagé
  - qu'on peut acquérir;
  - qui met en attente ceux qui le demandent;
  - qui donne la main dans l'ordre des demandes.

- Toutes les threads en **concurrence sur une ressource** partagent **un même sémaphore**
  - on **acquiert** le sémaphore avant d'entrer en **SC**;
  - on **relâche** le sémaphore en sortant de la **SC**.

IMPLÉMENTATION DU SÉMAPHORE



3 primitives (opérations non interruptibles = atomiques) :

“ init(val) ”	acquire()	release()

`compteur := val`
`file := vide`
	`compteur := compteur – 1`
	`si compteur < 0`
	`  bloque processus_courant`
	`  ajoute processus_courant à file`
		`compteur := compteur + 1`
		`si compteur ≤ 0`
		`  retire un processus de file`
		`  et le libère`

initialisation	*prend une ressource, attente si nécessaire*	*libère/produit une ressource*

UTILISATION DU SÉMAPHORE

```
1 Semaphore s = new Semaphore();
2 // ... code non-critique ...
3 s.acquire();
4 // ... code critique ...
5 s.release();
6 // ... code non-critique ...
```

Les propriétés d'un sémaphore

- **Vivacité** : on veut passer la main dans le bon ordre
  - ➠ utilisation d'une file d'attente des **PCB**
- Les fonctions `acquire` et `release` doivent être **atomiques** !
  - ➠ elles sont elles-même des **SC** pour le sémaphore ...

UTILISATION DU SÉMAPHORE

L'EXCLUSION MUTUELLE

Un sémaphore permettant de faire de l'**exclusion mutuelle**
est un sémaphore dont la valeur initiale est 1.

```
1 int cpt = 0;
2 Semaphore mutex = new Semaphore(1);
```

```
1 void Processus_1(){
2     mutex.acquire();
3     cpt = cpt + 1;
4     mutex.release();
5 }
```

```
1 void Processus_2(){
2     mutex.acquire();
3     cpt = cpt - 1;
4     mutex.release();
5 }
```

UTILISATION DU SÉMAPHORE

DEMI RENDEZ-VOUS

Quand un processus veut s'assurer qu'un traitement a été réalisé par un autre processus avant de réaliser le sien.

```
1 Semaphore sem = new Semaphore(0);
```

```
1 void Processus_1(){
2     traitement_1();
3     sem.release();
4 }
```

```
1 void Processus_2(){
2     sem.acquire();
3     traitement_2();
4 }
```

UTILISATION DU SÉMAPHORE

RENDEZ-VOUS

Permet à deux processus de définir un point de synchronisation.

```
1 Semaphore sem1 = new Semaphore(0);
2 Semaphore sem2 = new Semaphore(0);
```

```
1 void Processus_1(){
2     traitement_11();
3     sem1.release();
4     sem2.acquire();
5     traitement_12();
6 }
```

```
1 void Processus_2(){
2     traitement_21();
3     sem2.release();
4     sem1.acquire();
5     traitement_22();
6 }
```

UTILISATION DU SÉMAPHORE

PARTAGE DE N RESSOURCES

Soient N **ressources** disponibles et P **processus** voulant avoir accès à au moins l'une de ces ressources ($P > N$).

```
1 Semaphore sem = new Semaphore(N);
```

```
1 void Processus_i(){
2     debutTraitement();
3     sem.acquire();
4     utiliserLaRessource();
5     sem.release();
6     finTraitement();
7 }
```

UTILISATION DU SÉMAPHORE

INTER-BLOCAGE (DEADLOCK)

```
1 Semaphore sem1 = new Semaphore(1);
2 Semaphore sem2 = new Semaphore(1);
```

```
1 void Processus_1(){
2     sem1.acquire();
3     sem2.acquire();
4     utiliserRessource1();
5     utiliserRessource2();
6     sem1.release();
7     sem2.release();
8 }
```

```
1 void Processus_2(){
2     sem2.acquire();
3     sem1.acquire();
4     utiliserRessource2();
5     utiliserRessource1();
6     sem1.release();
7     sem2.release();
8 }
```

✘ Le programmeur doit s'assurer qu'il ne crée pas d'**interblocage**

UTILISATION DU SÉMAPHORE

INTER-BLOCAGE (SOLUTION)

```
1 Semaphore sem1 = new Semaphore(1);
2 Semaphore sem2 = new Semaphore(1);
```

```
1 void Processus_1(){
2   sem1.acquire();
3   utiliserRessource1();
4   sem1.release();
5   sem2.acquire();
6   utiliserRessource2();
7   sem2.release();
8 }
```

```
1 void Processus_2(){
2   sem2.acquire();
3   utiliserRessource2();
4   sem2.release();
5   sem1.acquire();
6   utiliserRessource1();
7   sem1.release();
8 }
```

PLAN

Retour au plan - Retour à l'accueil

SYNTHÈSE

- Problème d'utilisation des **ressources partagées**
- **Exemple** : accès à une variable partagée
- Notion de **section critique**
- Propriétés d'**exclusion mutuelle**, de **déroulement** et de **vivacité**
- Problème de l'**attente active**
- **Moniteurs**
- **Sémaphores**

MERCI

Version PDF des slides

Retour à l'accueil - Retour au plan