

# LES SYSTÈMES D'EXPLOITATION

## SYNCHRONISATION DES PROCESSUS ET DES THREADS

🎓 3A - Coursus Ingénieurs - Dominante Informatique et Numérique

🏛️ CentraleSupélec - Université Paris-Saclay - 2025/2026



**Idir AIT SADOUNE**  
[idir.aitsadoune@centralesupelec.fr](mailto:idir.aitsadoune@centralesupelec.fr)

# PLAN

- Les sections critiques
- Les solutions programmées
- Les mécanismes de synchronisation
- La synthèse

[Retour au plan](#) - [Retour à l'accueil](#)

# INTER PROCESS COMMUNICATION - IPC

- L'**OS** garantit l'**indépendance des processus**
  - par l'**ordonnanceur CPU**
  - par la **gestion mémoire** que l'on verra plus tard
- Un processus peut **communiquer** avec d'autres processus ou avec des périphériques (fichiers, imprimantes, réseaux, ...).
- Il est nécessaire de mettre en oeuvre un des **mécanismes** qui permettent aux processus d'**échanger des données** et de **se synchroniser** dans un **OS**.
  - **Objectif** → coordonner l'exécution et partager l'information entre 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 DE VARIABLE PARTAGÉE

Soit la gestion d'un compte bancaire

- Une variable partagée **balance**
- Une fonction **add(1)** ( $\text{balance} = \text{balance} + 1$ )
- Une fonction **sub(1)** ( $\text{balance} = \text{balance} - 1$ )
- 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 DE VARIABLE PARTAGÉE

```
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

- Le premier thread exécute 10 000 000 fois add(1)
- Le deuxième thread exécute 10 000 000 fois sub(1)

➡ **Résultat attendu** → balance = 9€

✗ **Résultat obtenu** → balance = -98599€

# PROBLÈME DE LA CONCURRENCE

## EXEMPLE DE VARIABLE PARTAGÉE

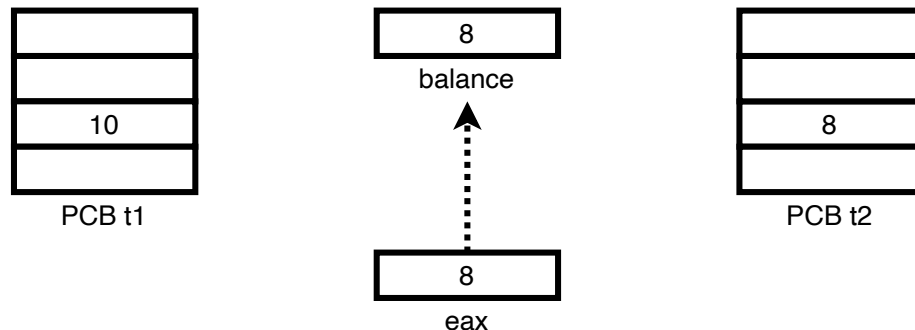
```
1 ;add(1) or balance = balance + 1
2 mov eax, balance
3 add eax, 1
4 mov balance, eax
```



```
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 → **balance = 8** alors qu'on s'attend à **balance = 9**

- **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

> Les sections critiques

> Les solutions programmées

> Les mécanismes de synchronisation

> La synthèse

[Retour au plan](#) - [Retour à l'accueil](#)

# SECTION CRITIQUE

- Une **section critique** désigne la **portion de code** dans laquelle un processus (ou un thread) accède à une **ressource partagée**.
  - ressource qui ne **doit pas être utilisée simultanément** par plusieurs processus.
  - **exemple** → une variable, un fichier, une structure de données en mémoire
- Une **section critique** est un **bloc de code** qui doit **s'exécuter de manière exclusive**
  - **un seul processus** à la fois peut y entrer.
  - si un processus est dans sa section critique, **les autres doivent attendre**.
- Sans mécanisme de protection → deux processus peuvent modifier la ressource en même temps → erreurs, données incohérentes.

```
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
```

# CONDITIONS DE LA SECTION CRITIQUE (SELON DIJKSTRA)

- Lorsqu'on déclare une **section critique**, on doit garantir qu'**au plus un seul** processus/thread est dans la section critique.
- Un bon mécanisme de gestion des sections critiques doit respecter :
  1. **Exclusion mutuelle** → un seul processus dans la section critique à la fois.
  2. **Progrès** → si aucun processus n'est dans la section critique, l'accès doit être possible.
  3. **Attente bornée (bounded waiting)** → tout processus voulant entrer finira par le faire (**pas d'attente infinie**).

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

# PLAN

- Les sections critiques
- Les solutions programmées
- Les mécanismes de synchronisation
- La synthèse

[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**

```
1 interface Mutex {  
2     abstract void commencerSC(int id);  
3     abstract void finirSC(int id);  
4     // id dans {0,1} num du thread  
5 }
```

```
1 //... code non critique  
2 my_mutex.commencerSC(my_id);  
3 //... code critique  
4 my_mutex.finirSC(my_id);  
5 //... 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

## PREMIÈRE IMPLÉMENTATION

```
1 interface Mutex {  
2     abstract void commencerSC(int id);  
3     abstract void finirSC(int id);  
4     // id dans {0,1} num du thread  
5 }
```

```
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

## PREMIÈRE IMPLÉMENTATION

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

id=0 →      → id=1

|           |      |
|-----------|------|
| est_SC[0] | true |
| est_SC[1] | true |

- ✗ Les deux threads sont en SC!
- ✗ La fonction `commencerSC` contient elle-même des SC!

# UNE SOLUTION POUR 2 THREADS

## DEUXIÈME IMPLÉMENTATION

```
1 interface Mutex {  
2     abstract void commencerSC(int id);  
3     abstract void finirSC(int id);  
4 }
```



```
1 //Mettre en attente le thread demandeur 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

## DEUXIÈME IMPLÉMENTATION

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

id=0   id=1

|           |      |
|-----------|------|
| est_SC[0] | true |
| est_SC[1] | true |

✗ **Problème** d'interblocage (**deadlock**) → processus bloqués indéfiniment

# UNE SOLUTION POUR 2 THREADS

## TROISIÈME IMPLÉMENTATION

```
1 interface Mutex {  
2     abstract void commencerSC(int id);  
3     abstract void finirSC(int id);  
4 }
```

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

# UNE SOLUTION POUR 2 THREADS

## 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 }
```

id=0 →

→ id=1

|           |      |
|-----------|------|
| est_SC[0] | true |
| est_SC[1] | true |
| tour      | 0    |

✓ Un seul thread est passé, l'autre est en attente

# UNE SOLUTION POUR 2 THREADS

## 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 }
```

✗ les processus font de l'**attente active** → utilisation inutile du processeur

✓ privilégier les **mécanismes de haut niveau** 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

- Les sections critiques
- Les solutions programmées
- Les mécanismes de synchronisation
- La synthèse

[Retour au plan](#) - [Retour à l'accueil](#)

# LE CONCEPT DE MONITEUR

Un **moniteur** est un module constitué de :

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

## Exemple de moniteurs en Java

- en **Java**, on peut déclarer que des **blocs de code** sont en **exclusion mutuelle**.
  - géré au niveau de la **JVM** (machine virtuelle de **Java**)
  - utilisation du mot clé **synchronized** → **un verrou (lock)** sur un objet
  - un seul thread dans les blocs **synchronized** sur le même objet
- un thread, qui possède un verrou, peut exécuter n'importe quelle méthode, verrouillée ou non (**verrou récursif**).
- un thread peut **verrouiller plusieurs objets** → risque **d'interblocage**.

# MONITEURS EN JAVA

## EXEMPLE D'UNE VARIABLE PARTAGÉE

```
1 public class Account {
2     // objet inaccessible de l'exterieur
3     private int value;
4
5     // pas de risque sur le constructeur
6     public Account(int i) {
7         this.value = i;
8     }
9
10    // fonctions manipulant l'objet en exclusion mutuelle
11    synchronized public void add(int v){
12        this.value = this.value + v;
13    }
14
15    synchronized public void sub(int v){
16        this.value = this.value - v;
17    }
18 }
```



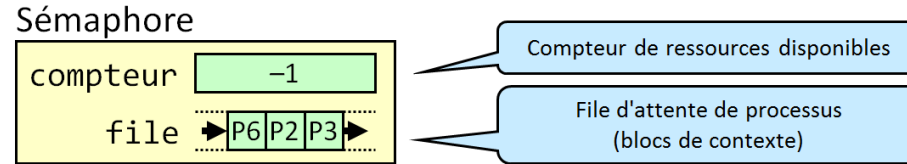
# SÉMAPHORE - DIJKSTRA - 1962

- Un **sémaphore** est un **mécanisme de synchronisation** utilisé pour **contrôler l'accès** à des ressources partagées par plusieurs processus/threads.
- **Un sémaphore** définit un objet **partagé** permettant de compter **le nombre de ressources disponibles**
  - qu'on peut **acquérir** (le **sémaphore**);
  - qui **met en attente** ceux qui le demandent s'il n'y a pas de ressources disponibles;
  - qui traite les demandes dans **l'ordre des arrivées**.
- Tous les threads en **concurrence sur une ressource** doivent partager **un même sémaphore**
  - on **acquiert** le sémaphore **avant d'entrer** en **SC** (l'appel système **Unix sem\_wait()**);
  - on **relâche** le sémaphore **en sortant** de la **SC** (l'appel système **Unix sem\_post()**).

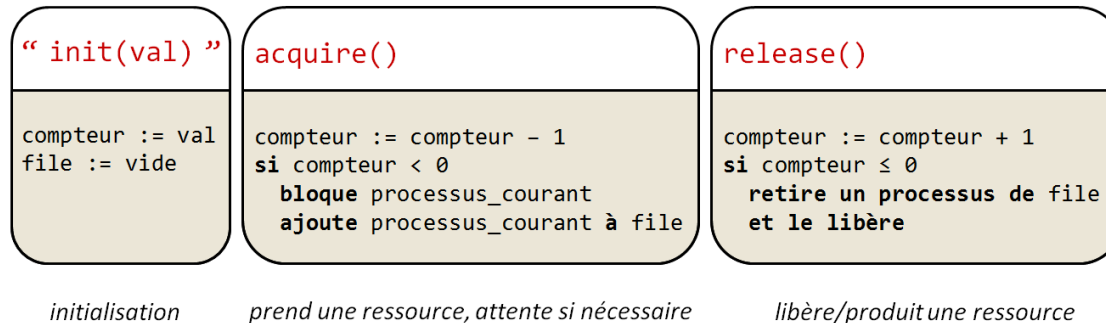
# SÉMAPHORE - DIJKSTRA - 1962

## IMPLÉMENTATION DU SÉMAPHORE

Un **sémaphore** est un compteur protégé qui régule l'accès concurrent à une ressource.

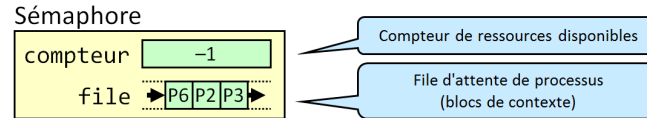


3 primitives (opérations **non interruptibles** = atomiques) :

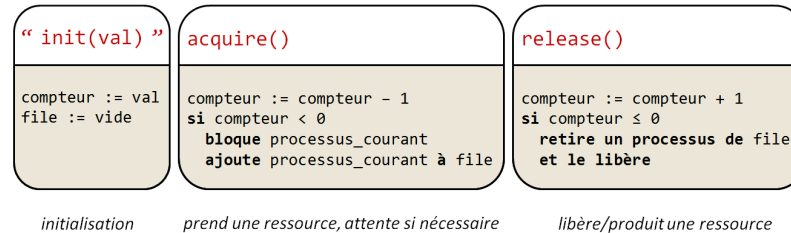


# SÉMAPHORE - DIJKSTRA - 1962

## LES PROPRIÉTÉS D'UN SÉMAPHORE



3 primitives (opérations **non interruptibles** = atomiques) :



- Les fonctions **acquiere** (**sem\_wait**)/**release** (**sem\_post**) doivent être **atomiques**
  - elles-même des **sections critiques** pour le sémaphore ...
  - l'**OS** est en charge de garantir cette propriété
- Vivacité** → on veut passer la main dans le bon ordre
  - utilisation d'une **file d'attente** des **PCB** géré par l'**OS**

# UTILISATION DU SÉMAPHORE

## SCHÉMA GÉNÉRAL

Schéma général de l'utilisation d'un **sémaphore**

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

# 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 avec  $P > n$ .

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

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



# L'INTERBLOCAGE (DEADLOCK)

L'**interblocage** est une situation dans laquelle **plusieurs processus** sont bloqués **indéfiniment** parce que chacun attend une ressource détenue par un autre.

Pour produire un **deadlock**, ces conditions doivent être vraies simultanément (**conditions de Coffman**) :

## 1. **Exclusion mutuelle**

- une ressource ne peut être utilisée que par un seul processus à la fois.

## 2. **Rétention et attente**

- un processus garde une ressource tout en attendant une autre.

## 3. **Non-préemption**

- une ressource ne peut pas être retirée de force à un processus.

## 4. **Attente circulaire**

- il existe une chaîne circulaire de processus où chacun attend une ressource détenue par le suivant.

# INTER-BLOPAGE (DEADLOCK)

## EXEMPLE

```
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**

# INTER-BLOCAGE (DEADLOCK)

## UNE SOLUTION POSSIBLE

```
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 }
```

- ✓ Chaque processus ou thread doit **acquérir** et **utiliser séparément les sémaphores** associés aux ressources partagées.

# PLAN

- Les sections critiques
- Les solutions programmées
- Les mécanismes de synchronisation
- La synthèse

[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](#)