

# Gestion des ressources

Jean-Michel Adam

UGA

# Les Ressources (vocabulaire)

## Ressources partageables simultanément

- code réentrant : conçu pour être exécuté simultanément par plusieurs processus
- fichier ou zone mémoire en lecture seule
- périphériques partageable (disques)

## Ressources banalisées

Ensemble de ressources de même type pouvant être utilisées indifféremment les unes des autres :

- tampons (buffers) mémoires
- dérouleurs de bande
- imprimantes (si identiques)

# Les Ressources

## Ressources réutilisables séquentiellement

- code non réentrant
- toute mémoire en lecture/écriture
- processeur
- périphériques séquentiels (imprimante, dérouleur de bande)

→ Ce sont des *ressources critiques*

## Ressources réquisitionnables ou non

## Ressources virtuelles (disque, mémoire, ...)

# Partage de ressource

Exemple : Partage d'une donnée en mémoire

- Deux tâches (A et B) s'exécutent simultanément
- Elles doivent modifier la même donnée
- Chaque tâche doit donc :
  - lire cette donnée,
  - calculer sa nouvelle valeur
  - mémoriser la nouvelle valeur

Problème : l'exécution simultané des tâches A et B peut perturber les calculs

# Partage de ressource

## Section critique

Lire

Modifier

Écrire

## Exemple : partage d'une variable Solde

Tâche A : dépôt de chèques

```
courA <- Solde;  
courA <- courA + mtCh  
Solde <- courA;
```

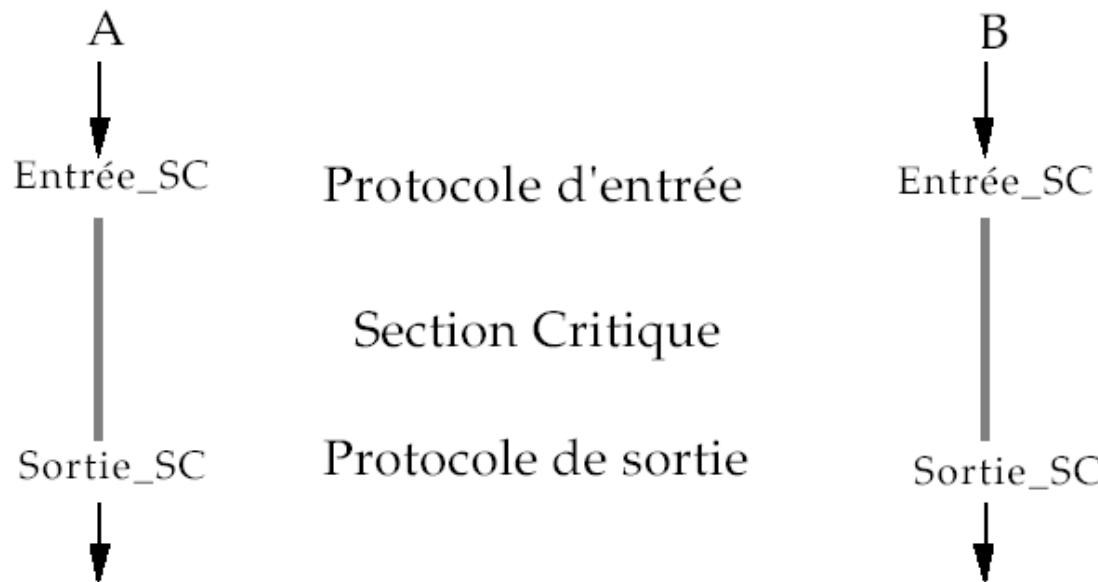
Tâche B : dépôt de liquide

```
courB <- Solde;  
courB <- courB + mtL  
Solde <- courB;
```

## Exécution sans protection :

Solde = X		courA		courB
X+mtL	courA <- Solde	X	courB <- Solde courB <- courB + mtL Solde <- courB	X X+mtL
X+mtCh	courA <- courA + mtCh Solde <- courA	X+mtCh		

# Exclusion mutuelle : éléments de base



1. Une section critique est occupée par une seule tâche à la fois : **deux processus ne peuvent être dans la même section critique**
2. Aucune hypothèse doit être faite sur la vitesse relative des tâches, ni sur le nombre de tâches
3. Aucune tâche suspendue en dehors d'une section critique ne doit empêcher une autre d'y entrer (pas d'interblocage)
4. Une tâche doit attendre un temps **fini** devant une section critique (pas de famine)

# Exclusion mutuelle : verrouillage par une variable

- Les processus partagent des variables pour synchroniser leurs actions
- Pour entrer dans une section critique, chaque processus doit consulter **une variable booléenne unique** (initialisée à faux) indiquant si la ressource critique est occupée
- Si elle vaut faux, le processus la met à vrai et entre en section critique
- Si le verrou est à vrai, le processus attend qu'elle soit à faux

# Attente active (1)

## Solution triviale

Tâche T1 :

```
...  
Entrée_SC()  
...  
Sortie_SC()  
...
```

Tâche T2 :

```
...  
Entrée_SC()  
...  
Sortie_SC()  
...
```

**Entrée\_SC( )**

```
tantque occupée faire  
début fin;
```

```
occupée <- VRAI
```

**Sortie\_SC( ) :**

```
occupée <- FAUX
```

Pb : exclusion mutuelle non assurée !

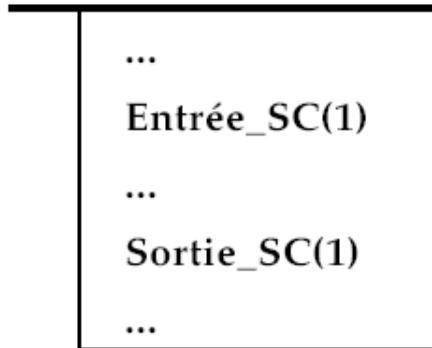
**Init\_SC( )**

```
occupée <- FAUX
```

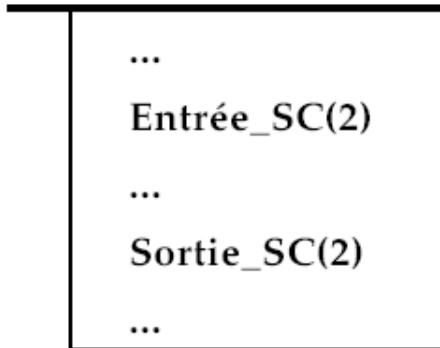
# Attente active (2)

Solution avec tableau de booléens solution avec un tableau de booléens qui indiquent qui veut entrer en section critique

Tâche T1 :



Tâche T2 :



Entrée\_SC(Consulté i : entier)  
    demande[i]  $\leftarrow$  VRAI  
    tantque demande[3-i] faire  
        ftq;

Sortie\_SC(Consulté i : entier )  
    demande[i]  $\leftarrow$  FAUX

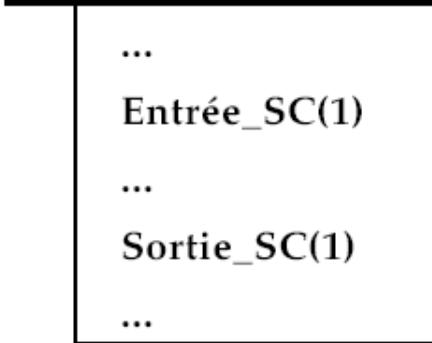
Init\_SC( )  
    demande[1]  $\leftarrow$  FAUX  
    demande[2]  $\leftarrow$  FAUX

Exclusion mutuelle assurée mais risque d'interblocage !

# Attente active (3)

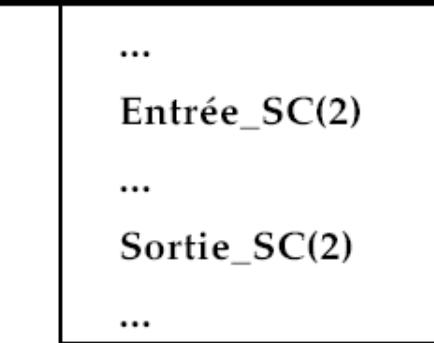
## Solution avec variable entière tour pour assurer l'alternance

Tâche T1 :



```
Entrée_SC(Consulté i : entier)
tantque (tour = 3-i) faire
début fin;
```

Tâche T2 :



```
Sortie_SC(Consulté i : entier )
tour ← 3-i
```

```
Init_SC( )
tour ← 1
```

Exclusion mutuelle assurée mais risque de famine !

# Attente active (4)

## Solution de Dekker-Peterson

Tâche T1 :

...  
Entrée\_SC(1)

...  
Sortie\_SC(1)

...

Tâche T2 :

...  
Entrée\_SC(2)

...  
Sortie\_SC(2)

...

Entrée\_SC(Consulté i : entier)

    demande[i]  $\leftarrow$  vrai

    tour  $\leftarrow$  3-i

    tantque demande[3-i] et tour  $\neq$  i

        début fin;

Sortie\_SC(Consulté i : entier )

    demande[i]  $\leftarrow$  FAUX

Init\_SC( )

    demande[1]  $\leftarrow$  FAUX

    demande[2]  $\leftarrow$  FAUX

# Mécanismes matériels pour l'exclusion mutuelle

- Mono-processeur : masquage des interruptions  
Entrée\_SC : masquage  
Sortie\_SC : démasquage

# Mécanismes matériels

- Multiprocesseurs : Instruction Test and Set Lock

fonction Test\_And\_Set (Modifié occupée : booléen) → booléen  
booléen TAS

TAS ← occupée  
occupée ← VRAI  
renvoyer(TAS)

Attente active en configuration multiprocesseurs avec Test and Set Lock

Entrée\_SC( )

---

Test :    tantque Test\_and\_Set(occupée)  
              fintantque

---

Sortie\_SC( )

---

occupée <- FAUX;

---

Init\_SC( )

---

occupée <- FAUX;

---

# Solution à base d'attente passive

Problème des solution précédentes :

- Gaspillage de l'UC (attente active)

Principe des solutions à base d'attente passive :

- Endormir un processus lorsque la section est verrouillée
- Le réveiller lorsqu'elle se libère

→ utilisation de **verrous**

# Classe Verrou

**Classe verrou**

**privé** ouvert : booléen;

**privé** attente : liste de tâches;

**public procédure** verrouiller()

**début**

**si** ouvert **alors**

      ouvert  $\leftarrow$  faux

**sinon**

*Ajouter la tâche dans la liste attente*

*dormir*

**finsi;**

**fin** verrouiller;

**fin** verrou;

**public procédure** init()

**début**

    ouvert  $\leftarrow$  vrai;

    attente  $\leftarrow$  vide;

**fin** init

**public procédure** déverrouiller()

**début**

**si** attente  $\neq$  vide **alors**

*Sortir la première tâche de attente,*

*La réveiller*

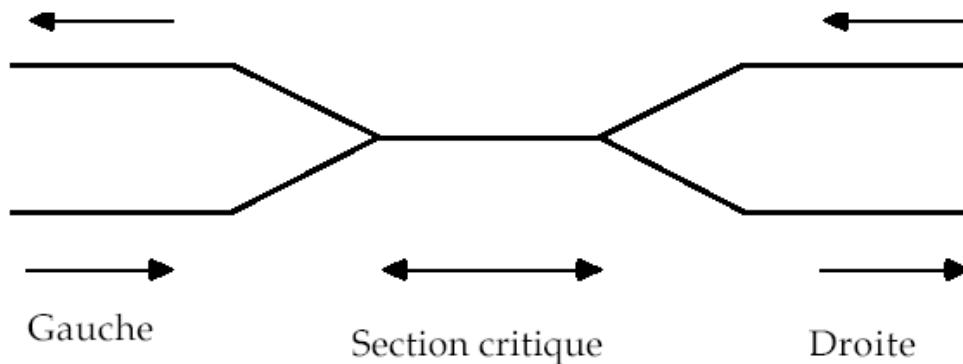
**sinon**

      ouvert  $\leftarrow$  vrai

**finsi;**

**fin** déverrouiller;

# voie unique avec classe Verrou



**Tâche TrainGauche (voie : verrou);  
début**

```
rouler;
voie.verrouiller();
Passer;
voie.déverrouiller();
rouler;
fin TrainGauche;
```

**Tâche TrainDroit (voie : verrou);  
début**

```
rouler;
voie.verrouiller();
Passer;
voie.déverrouiller();
rouler;
fin TrainDroit;
```

# Sémaphores

(Dijkstra)

- Mécanisme de synchronisation entre processus.
- Un **sémaphore S** est une variable entière, manipulable par l'intermédiaire de deux opérations :  
**P** (*proberen*) et **V** (*verhogen*)  
acquire              release

**acquire (S)**         $S \leftarrow (S - 1)$   
*si  $S < 0$ , alors mettre le processus en attente ;*  
*sinon on poursuit l'exécution*

**release (S)**         $S \leftarrow (S + 1)$  ; *réveil d'un processus en attente.*

- **acquire (S)** correspond à une tentative de franchissement. S'il n'y a pas de jeton pour la section critique alors attendre, sinon prendre un jeton et entrer dans la section (S), puis rendre son jeton à la sortie de la section critique.
- Chaque dépôt de jeton **release (S)** autorise un passage. Il est possible de déposer des jetons à l'avance

# Classe Séaphore

**Classe Séaphore**

**privé** val : entier;

**privé** attente : liste de tâches;

**public procédure** init\_sémaphore(**consulté** v : entier);

**début**

val  $\leftarrow$  v

attente  $\leftarrow$  vide

**fin** init\_sémaphore

**public procédure** acquire ()

**début**

val  $\leftarrow$  val - 1

**si** val < 0 **alors**

*Ajouter la tâche dans la liste attente*

*dormir*

**finsi**

**fin** acquire

**fin** Séaphore

**public procédure** release ()

**début**

val  $\leftarrow$  val + 1;

**si** val  $\leq$  0 **alors**

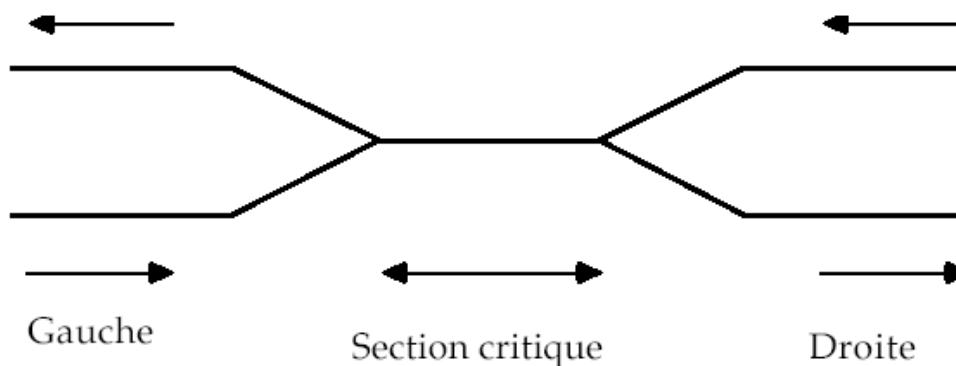
*Sortir la première tâche de attente*

*La réveiller*

**finsi**;

**fin** release ;

# Exemple : voie unique



**Tâche TrainGauche (modifié voie : sémaphore);  
début**

```
rouler;  
voie.acquire();  
Passer;  
voie.release();  
rouler;  
fin TrainGauche;
```

```
voie.init_sémaphore(1)
```

**Tâche TrainDroit (modifié voie : sémaphore);  
début**

```
rouler;  
voie.acquire();  
Passer;  
voie.release();  
rouler;  
fin TrainDroit;
```

# Difficultés liées au partage de ressources

## Interblocage

### Exemple avec des sémaphores

#### Tâche A

```
mutex1.acquire() (1a)  
mutex2.acquire() (2a)  
...  
mutex2.release()  
mutex1.release()
```

#### Tâche B

```
mutex2.acquire() (1b)  
mutex1.acquire() (2b)  
...  
mutex1.release()  
mutex2.release()
```

L'ordre d'exécution (1a) (1b) (2a) (2b) crée un interblocage (*deadlock*)

# Interblocage (*deadlock*)

- **Conditions :**
  1. Des ressources non partageables
  2. Des processus qui conservent des ressources déjà obtenues alors qu'ils sont en attentes d'autres ressources
  3. Les demandes de ressources sont bloquantes et il n'y a pas de préemption (les ressources ne peuvent être réquisitionnées)
  4. Il existe une chaîne circulaire de processus telle que chacun réclame les ressources possédées par le suivant
- **Solutions :**
  - Prévention (en supprimant une des conditions)
  - Détection
  - Guérison

# Solutions au problème de l'interblocage

## Prévention

Allocation de ressources de manière « intelligente »

## Prévention a priori

- **Méthode de l'allocation globale des ressources à une tâche (cond. 2)**
  - => immobilisation non productive de ressources
  - => perte d'une bonne partie du parallélisme
- **Méthode des classes ordonnées (cond. 4)**
  - Les ressources sont partagées en classe C0...CM et on n'alloue les ressources de la classe k que si la tâche a déjà obtenu les ressources de la classe k-1.
  - L'ordre d'allocation est le même pour toutes les tâches, on élimine ainsi la condition 4 (cycle dans les besoins et les allocations).

# Solutions au problème de l'interblocage

## Algorithme du banquier

- Annonce des besoins
- Réévaluation du risque à chaque allocation (complètement centralisée)
- Fondé sur la notion d'*état fiable* : état à partir duquel on peut s'exécuter sans interblocage même dans l'hypothèse la plus pessimiste.

## Détection

- On peut utiliser pour détecter l'interblocage l'algorithme qui détermine si un état est fiable ou non.

## Guérison

- Il faut reprendre l'exécution dans un état fiable, deux méthodes principales :
  1. détruire les tâches interbloquées l'une après l'autre, jusqu'à obtenir un état fiable
  2. sauvegarder périodiquement l'état des tâches quand on est dans un état fiable => points de reprise.

# Solutions au problème de l'interblocage

## **Que fait-on en pratique ?**

- Ni détection, ni guérison : on organise le système de façon à limiter au maximum les risques et on renvoie le programmeur à ses responsabilités.
- Tous les appels systèmes susceptibles de bloquer une tâche sont implémentés avec des délais d'attente au delà desquels l'appel retourne un code d'erreur.

