Gestion des ressources

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 : dépot de chèques

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

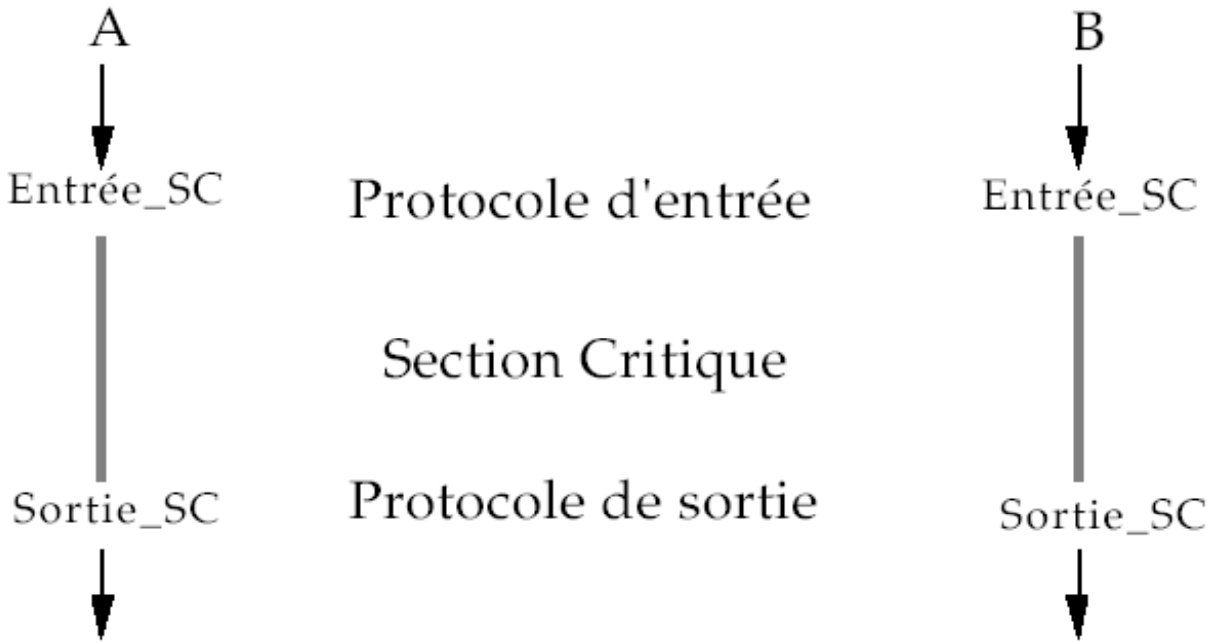
Tâche B : dépôt de liquide

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

Exécution sans protection :

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

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
```

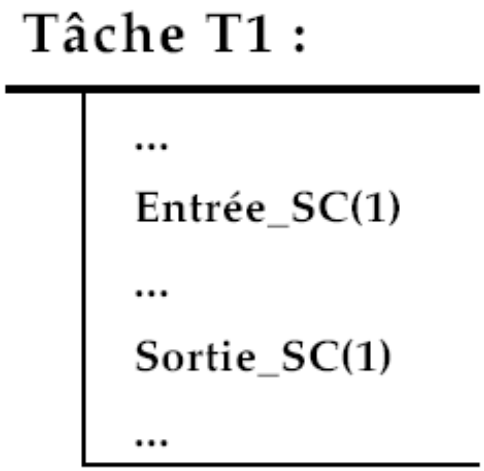
Pb : exclusion mutuelle non assurée !

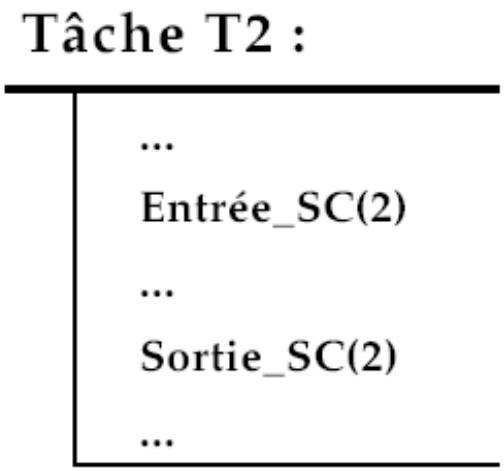
Init_SC( )

```
occupée <- FAUX
```

Attente active (2)

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






```
Entrée_SC(Consulté i : entier)
        demande[i] ← VRAI
        tantque demande[3-i] faire
        début fin;
```

```
Sortie_SC(Consulté i : entier )
        demande[i] ← FAUX
```

```
Init_SC( )
        demande[1] ← FAUX
        demande[2] ← FAUX
```

Exclusion mutuelle assurée mais risque d'interblocage !

Attente active (3)

Solution avec variable entière tour pour assurer l'alternance

Tâche T1 :

```
...
Entrée_SC(1)
...
Sortie_SC(1)
...
```

Tâche T2 :

```
...
Entrée_SC(2)
...
Sortie_SC(2)
...
```

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

```
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] ← vrai
        tour ← 3-i
        tantque demande[3-i] et tour ≠ i
        début fin;
```

```
Sortie_SC(Consulté i : entier )
        demande[i] ← FAUX
```

```
Init_SC( )
        demande[1] ← FAUX
        demande[2] ← 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

$\rightarrow$ utilisation des verrous

Verrous

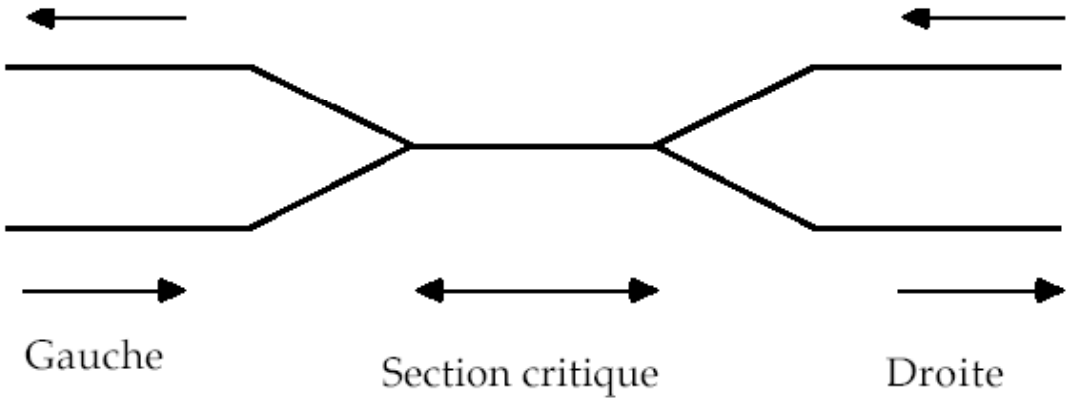
```
Type verrou = structure
            ouvert : booléen;
            attente : liste de tâches;
fin verrou;
```

```
procédure init_verrou (modifié v : verrou);
début
        v.ouvert ← vrai;
        v.attente ← vide;
fin init_verrou;
```

```
procédure verrouiller (modifié v : verrou);
début
  si v.ouvert alors
     v.ouvert ← faux
  sinon
     Ajouter la tâche dans la liste v.attente
     dormir
 finsi;
fin verrouiller;
```

```
procédure déverrouiller (modifié v : verrou);
début
si v.attente ≠ vide alors
     Sortir la première tâche de v.attente,
     La réveiller
sinon
     v.ouvert ← vrai
finsi;
fin déverrouiller;
```

Exemple : voie unique



Tâche TrainGauche (**modifié** voie : verrou);
début
rouler;
verrouiller(voie);
Passer;
déverrouiller(voie);
rouler;
fin TrainGauche;

Tâche TrainDroit (**modifié** voie : verrou);
début
rouler;
verrouiller(voie);
Passer;
déverrouiller(voie);
rouler;
fin TrainDroit;

Sémaphores

- 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 ← (S − 1)*
  si S < 0, alors mettre le processus en attente ;
  sinon on poursuit l'exécution

  release (S) *S ← (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

Sémaphores

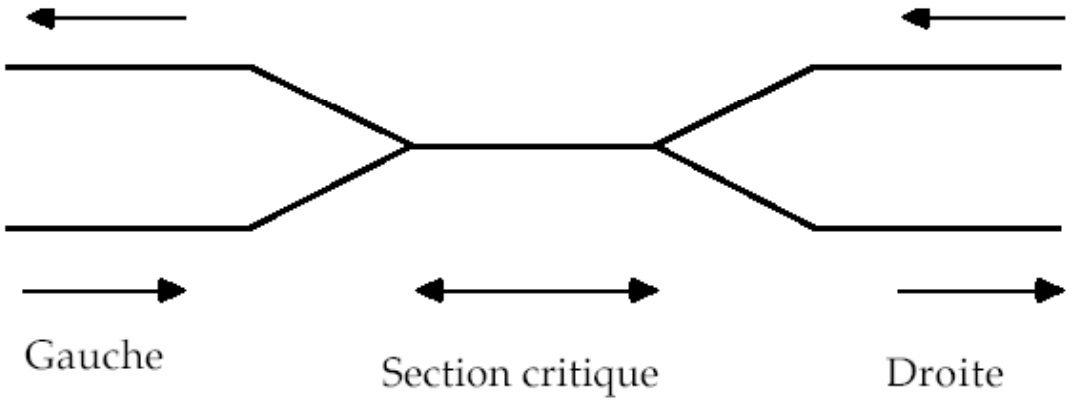
```
Type sémaphore = structure
   val : entier;
   attente : liste de tâches;
fin sémaphore;
```

```
procédure init_sémaphore
      ( modifié s : sémaphore;
        consulté valeur : entier);
début
    s.val ← valeur
    s.attente ← vide
fin init_sémaphore
```

```
procédure acquire (modifié s : sémaphore)
    s. val ← s.val - 1
    si s.val < 0 alors
          Ajouter la tâche dans la liste s.attente
          dormir
    finsi
fin P
```

```
procédure release (modifié s : sémaphore);
début
    s. val ← s.val + 1;
    si s.val ≤ 0 alors
          Sortir la première tâche de s.attente
          La réveiller
    finsi;
fin V;
```

Exemple : voie unique



Tâche TrainGauche (**modifié** voie : sémaphore);
début
rouler;
acquire(voie);
Passer;
release(voie);
rouler;
fin TrainGauche;

init_sémaphore(voie,1)

Tâche TrainDroit (**modifié** voie : sémaphore);
début
rouler;
acquire(voie);
Passer;
release(voie);
rouler;
fin TrainDroit;

Difficultés liées au partage de ressources

Interblocage : exemple avec des sémaphores

```
Tâche A :                          Tâche B :
P(mutex1);    (1a)                 P(mutex2);    (1b)
P(mutex2);    (2a)                 P(mutex1);    (2b)
...                                ...
V(mutex2);                         V(mutex1);
V(mutex1);                         V(mutex2);
```

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 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 (borne supérieure pour une ressource)
- 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.