Chapitre 6

La gestion des ressources

Section critique

Exclusion mutuelle

TSL

Les sémaphores
- Présentation
- Les primitives

Utilisation des sémaphores

Sémaphores : problèmes classiques

La gestion des ressources

- Le partage d'une ressource (mémoire, imprimante, ...) entre plusieurs processus peut entraîner des résultats incorrects.

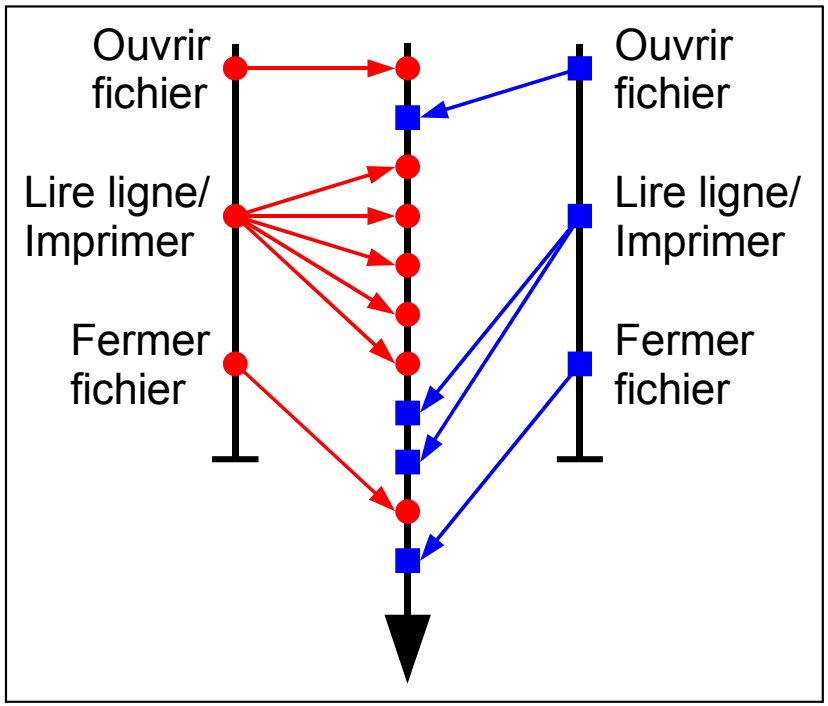
```
début
    ouvrir le fichier
    tant que lire(ligne)=vrai faire          // SECTION
        écrire la ligne sur l'imprimante     // CRITIQUE
    fermer le fichier
```

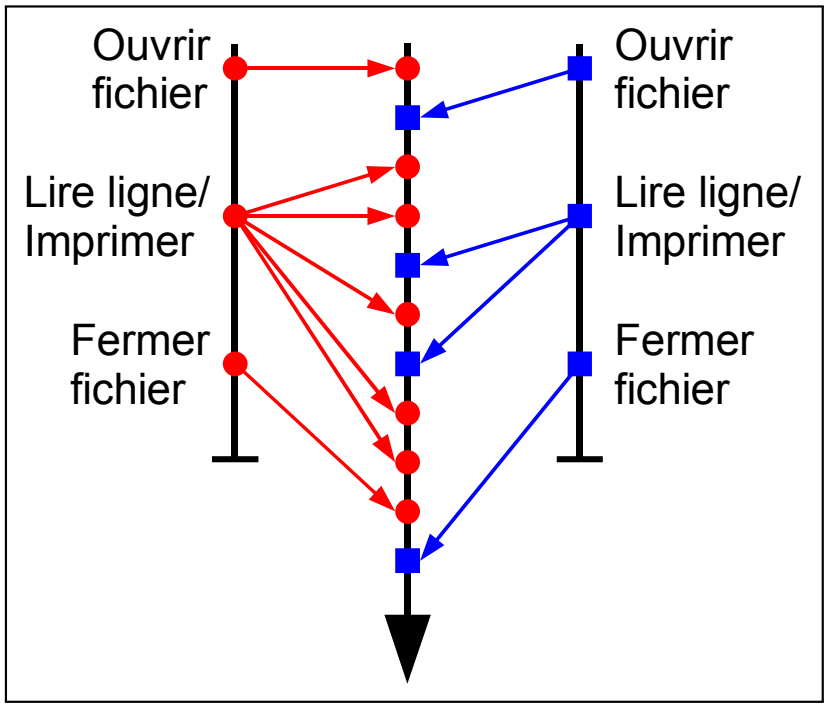
- Deux exécutions concurrentes de ce même programme impriment un listing contenant une imbrication des deux fichiers !

- **Définition :** une section critique est une partie de code ne devant être exécutée que par un seul processus à la fois.

La gestion des ressources : exemple

- Sur un système multi-tâches, on ne peut faire aucune hypothèse sur l'ordre d'exécution des instructions.





L'exclusion mutuelle

Pour que deux processus puissent **coopérer** efficacement pour le partage d'une ressource, il est nécessaire qu'ils respectent les contraintes suivantes :

1. deux processus ne peuvent pas être en même temps dans la section critique
2. aucune hypothèse n'est faite sur les vitesses relatives des processus et sur le nombre de processeurs
3. aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres
4. aucun processus ne doit attendre trop longtemps avant d'entrer dans la section critique

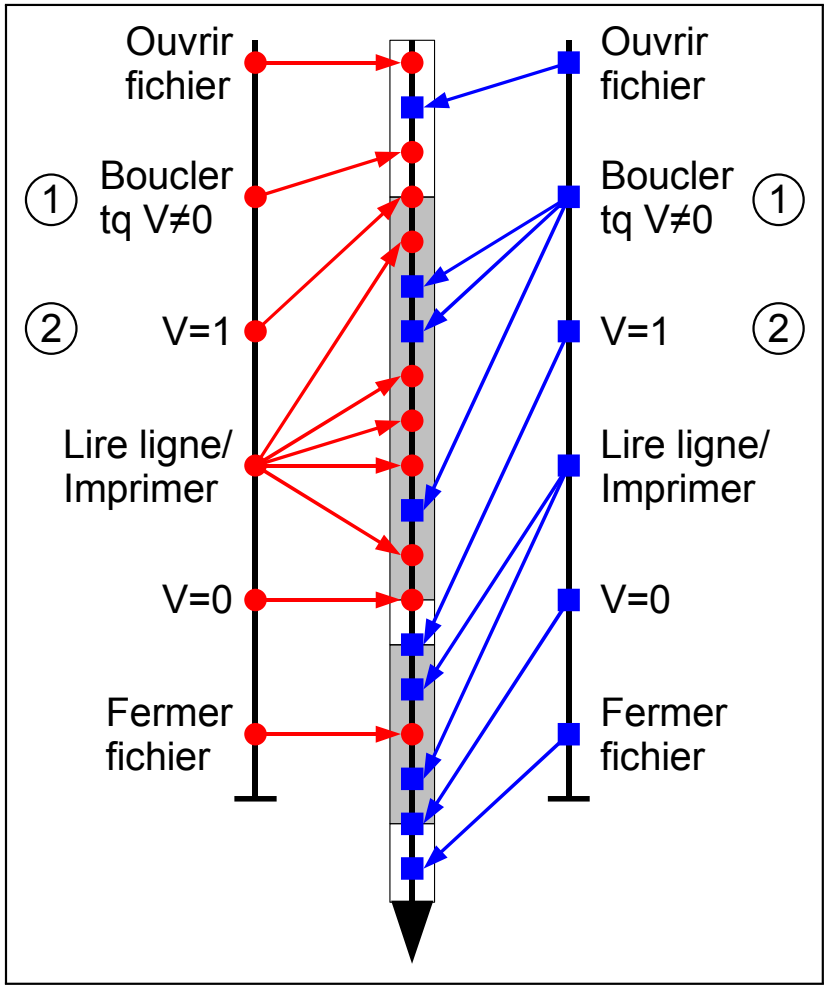
Fausse solution pour l'exclusion mututelle

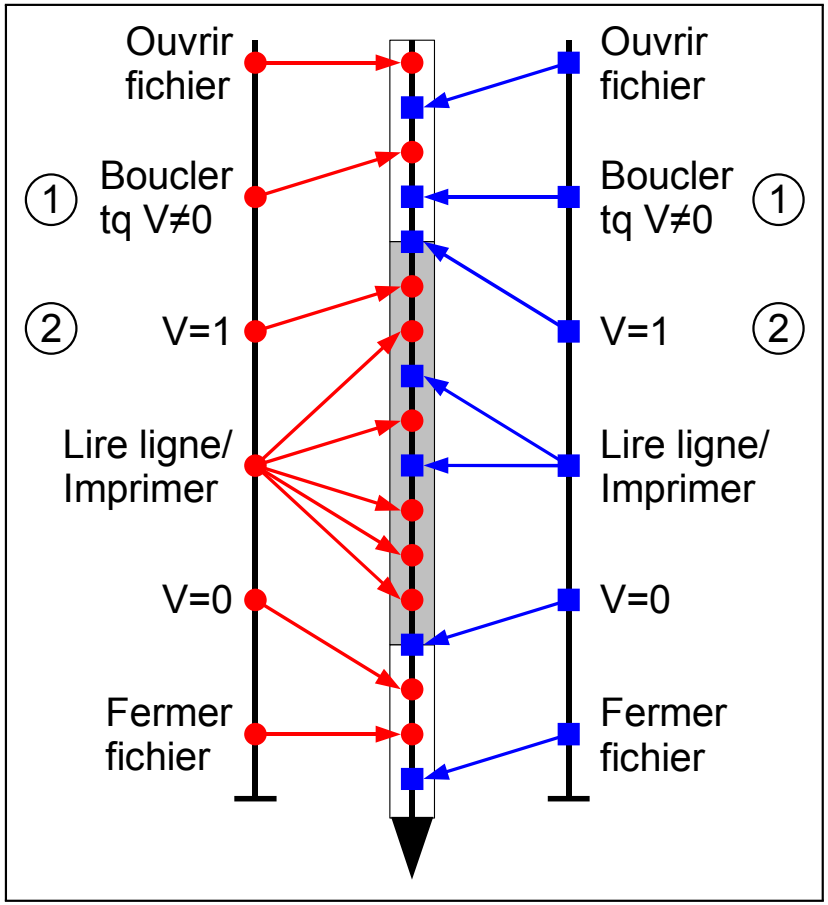
- La solution suivante *ne* résout *pas* le problème de l'exclusion mutuelle : elle ne fait que déplacer la section critique.

```
   début
       ouvrir le fichier
1      boucler tant que V ≠ 0        // SECTION
2      V = 1                         // CRITIQUE
       tant que lire(ligne)=vrai faire
           écrire la ligne sur l'imprimante
       V = 0
       fermer le fichier
```

- 2 processus peuvent effectuer la ligne (1) et trouver $V = 0$ car l'ordonnanceur peut interrompre un processus après l'exécution de la ligne (1) et avant l'exécution de la ligne (2).

Fausse solution pour l'exclusion mututelle : illustration





- Il faudrait que chaque processus exécute de manière indivisible les étapes (1) et (2).

L'instruction TSL (Test and Set Lock)

- De nombreux processeurs disposent d'une instruction

```
TSL(reg, adr)
```

- L'instruction `TSL` réalise de manière **indivisible** (ou atomique) les 2 opérations suivantes :
  1. charger le contenu de `adr` dans `reg` (`mov reg, adr`)
  2. mettre une valeur non nulle dans `adr` (`mov adr, #1`)

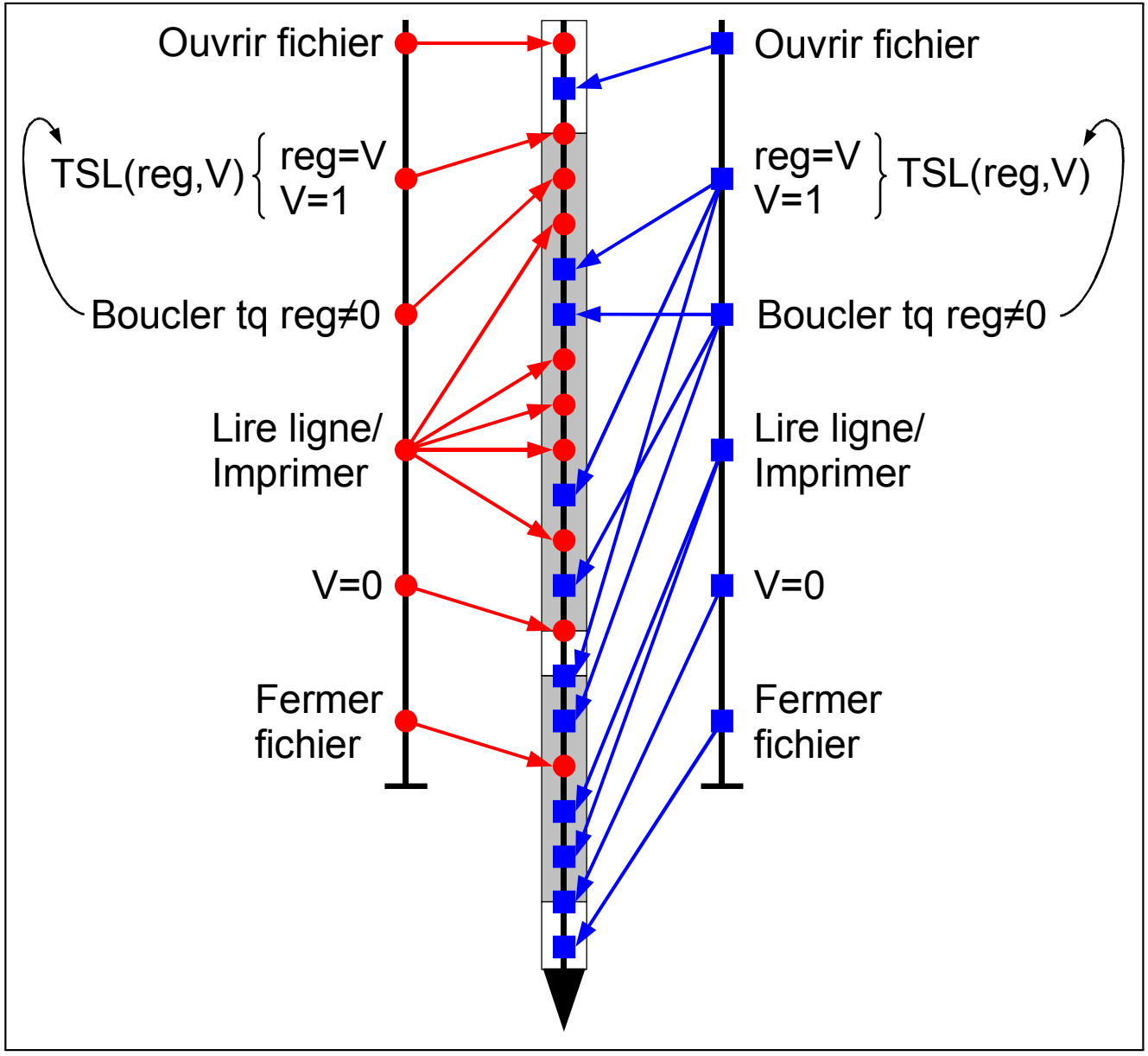
L'instruction TSL et l'exclusion mutuelle

- L'utilisation de l'instruction `TSL` permet de réaliser une exclusion mutuelle en utilisant :
  - une variable *partagée V* initialisée à 0
  - une variable *locale* `reg`

```
   début
       ouvrir le fichier
1      faire TSL(reg,V)
2      tant que reg ≠ 0
       tant que lire(ligne)=vrai faire
           écrire la ligne sur l'imprimante
       V = 0
       fermer le fichier
```

- Cette solution fonctionne MAIS entraîne une très grande consommation CPU (attente active des processus)

L'instruction TSL : illustration



Les sémaphores de Dijkstra (1965)

Un sémaphore peut être vu comme une variable partagée dont la valeur

- représente le nombre de processus pouvant encore utiliser la ressource
- est **toujours** positive ou nulle
- est parfois appelée *nombre de jetons*

Les 4 primitives des sémaphores

Quatre primitives permettent de manipuler les sémaphores.

- **P(sem)** ou down(sem)
  - Le processus réalisant cette opération teste la valeur de *sem*
  - Si *sem* = 0 alors le processus s'endort
  - Sinon *sem* est décrémentée
- **V(sem)** ou up(sem)
  - S'il existe des processus bloqués sur un P alors un de ces processus est réveillé
  - Sinon la valeur de *sem* est incrémentée
- **Z(sem)**
  - Bloque le processus tant que la valeur de *sem* n'est pas 0
- **R(sem)**
  - Retourne la valeur de *sem*

Algorithmes des opérations sur les sémaphores

En interne, est associé à chaque sémaphore :

- 2 variables partagées :
  - *jetons* : contient la valeur du sémaphore (c'est-à-dire le nombre de jetons)
  - *verrou* : est utilisé dans une instruction TSL avant d'accéder à *jetons*
- 2 listes :
  - *LP* : la liste des processus bloqués par une opération P sur ce sémaphore
  - *LZ* : la liste des processus bloqués par une opération Z sur ce sémaphore

Algorithme de P

```
début
    faire TSL(reg,sem.verrou) tant que reg ≠ 0
    si sem.jetons = 0 alors
        ajouter le processus dans LP
        sem.verrou = 0
        s'endormir
    sinon
        sem.jetons = sem.jetons - 1
        si sem.jetons=0 alors
            réveiller tous les processus de LZ
            vider LZ
        sem.verrou = 0 ;
```

Algorithme 1: P(sem)

Algorithme de V

```
début
    faire TSL(reg,sem.verrou) tant que reg ≠ 0
    si LP n'est pas vide alors
        retirer le premier processus de LP
        le réveiller
    sinon
        sem.jetons = sem.jetons + 1
    sem.verrou = 0 ;
```

Algorithme 2: V(sem)

Algorithme de Z

```
début
    faire TSL(reg,sem.verrou) tant que reg ≠ 0
    si sem.jetons ≠ 0 alors
        ajouter le processus dans LZ
        sem.verrou = 0
        s'endormir
    sinon
        sem.verrou = 0 ;
```

Algorithme 3: Z(sem)

Algorithme de R

début
  faire TSL(reg,sem.verrou) **tant que** reg $\neq$ 0
  tmp = sem.jetons
  sem.verrou = 0
  retourner *tmp*

Algorithme 4: R(sem)

Exclusion mutuelle avec les sémaphores

Complétez le code ci-dessous pour gérer l'accès à la section critique à l'aide d'un sémaphore (vous devrez toujours préciser le nombre initial de jetons).

début

traitement réalisé avant l'utilisation de la ressource

utilisation de la ressource non partageable

traitement réalisé après l'utilisation de la ressource

Rendez-vous avec les sémaphores (1)

- Les sémaphores permettent également de réaliser simplement des rendez-vous !
- **Première version** : on souhaite que le processus P_1 attente à la fin de son traitement qui dure 10 secondes le processus P_2 réalisant un traitement d'une durée de 30 secondes. Complétez le code ci-dessous.

Processus P_1

```
début
    répéter
        Traitement P1 (10 sec.)
    jusqu'à l'infini;
```

Processus P_2

```
début
    répéter
        Traitement P2 (30 sec.)
    jusqu'à l'infini;
```

Rendez-vous avec les sémaphores (2)

- **Deuxième version** : cette fois-çi les processus P_1 et P_2 doivent aussi se synchroniser mais le traitement de P_1 peut être plus ou moins long que celui de P_2. Complétez.

Processus P_1

début

répéter

Traitement P_1

jusqu'à *l'infini*;

Processus P_2

début

répéter

Traitement P_2

jusqu'à *l'infini*;

Rendez-vous à N avec les sémaphores

- Dans cette dernière version, *N* processus doivent se donner rendez-vous (*N* est connu avant l'exécution des processus).

Processus P_i

début

Avant rendez-vous

Après rendez-vous

Le problème de l'émetteur et du récepteur

Un processus émetteur dépose, à intervalle variable, un octet dans une variable à destination d'un processus récepteur. Quelle solution proposez-vous pour que l'émetteur ne dépose pas un nouvel octet alors que le récepteur n'a pas encore lu le précédent et que le récepteur ne lise pas deux fois le même octet.

Le problème des producteurs et des consommateurs

- Des processus producteurs produisent des objets et les insère un par un dans un tampon de *n* places. Bien entendu des processus consommateurs retirent, de temps en temps les objets (un par un).
- Résolvez le problème pour qu'aucun objet ne soit ni perdu ni consommé plusieurs fois.

Le problème des lecteurs et des rédacteurs

Ce problème modélise les accès à une base de données. On peut accepter que plusieurs processus lisent la base en même temps mais si un processus est en train de la modifier, aucun processus, pas même un lecteur, ne doit être autorisé à y accéder. Comment programmer les lecteurs et les rédacteurs ?