

Processus

Problèmes élémentaires de synchronisation

Vania Marangozova
Université Grenoble Alpes
2023-2024

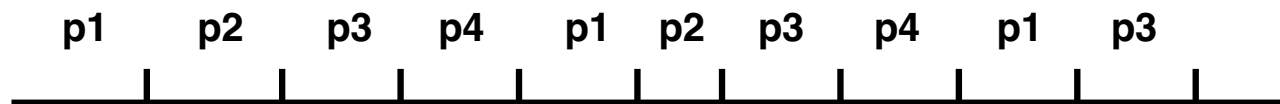
vania.marangozova@imag.fr

Cours basé sur les transparents de
Sacha Krakowiak et Renaud Lachaize



Raisonner sur les processus parallèles

- ▶ Dans un système d'exploitation, le processeur est multiplexé entre les processus prêts (quantum d'exécution)



- ▶ **Les instants de commutation sont indépendants de la logique interne du déroulement de chaque processus.**
- ▶ S'il y avait suffisamment de processeurs, il n'y aurait pas de partage (pas de commutation)
- ▶ Pour le raisonnement logique sur les processus,
il ne faut faire aucune hypothèse sur l'ordre relatif des exécutions
- ▶ Seuls comptent
 - **l'ordre d'exécution interne** à chaque processus
 - **les relations logiques** entre les processus (synchronisation)

Synchronisation entre processus

- ▶ Dans les deux cas de relations entre processus (compétition ou coopération), on doit **faire attendre** un processus. Comment réaliser cette attente ?

- ▶ **Solution 1 : attente active**

```

p1
while (ressource occupée)
{ };
ressource occupée = true;
...
    
```

```

p2
ressource occupée = true;
utiliser ressource;
ressource occupée = false;
    
```

- ▶ très peu économique si pseudo-parallélisme
- ▶ difficulté d'une solution correcte (à voir plus tard)

- ▶ **Solution 2 : blocage du processus**

- ▶ L'exécution d'un processus **bloqué** est arrêtée, jusqu'à son réveil explicite par un autre processus ou par le système

Le problème de l'exclusion mutuelle

► Opérations sur un compte bancaire

- les processus p1 et p2 sont lancés depuis deux agences différentes

processus p1

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 400
3. ecrire_compte (1867A, nouveau)
```

processus p2

```
1. courant = lire_compte (1867A)
2. nouveau = courant - 400
3. ecrire_compte (1867A, nouveau)
```

Le problème de l'exclusion mutuelle

► Opérations sur un compte bancaire

- les processus p1 et p2 sont lancés depuis deux agences différentes

processus p1

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

processus p2

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

► À noter...

- les variables courant et nouveau sont locales à chaque processus (elles sont dans sa mémoire virtuelle) : il y en a donc deux exemplaires distincts et indépendants
- les deux processus se déroulent en parallèle. L'exécution des opérations peut être entrelacée dans un ordre quelconque, à condition de respecter l'ordre local pour chacun des processus

Le problème de l'exclusion mutuelle

► Opérations sur un compte bancaire

- les processus p1 et p2 sont lancés depuis deux agences différentes

processus p1

```
1. courant = lire_compte (1867A)
2. nouveau = courant - 1000
3. ecrire_compte (1867A, nouveau)
```

processus p2

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

■ Exemples d'exécution

- **exécution 1 : p2.1 ; p2.2 ; p2.3 ; p1.1 ; p1.2 ; p1.3**
- **exécution 2 : p1.1 ; p1.2 ; p2.1 ; p2.2 ; p2.3 ; p1.3**
- **quels sont les résultats ? que peut-on en conclure ?**

Sections critiques et actions atomiques

Comment éviter les problèmes d'accès concurrent aux variables partagées ?

Assurer que l'ensemble des opérations (consultation + mise à jour) est exécutée de manière **indivisible** (atomique)

Pas d'interférences possibles de la part d'autres opérations exécutées en parallèle

processus p1

processus p2

A1

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

A2

```
1. courant = lire_compte (1867A)
2. nouveau = courant + 3000
3. ecrire_compte (1867A, nouveau)
```

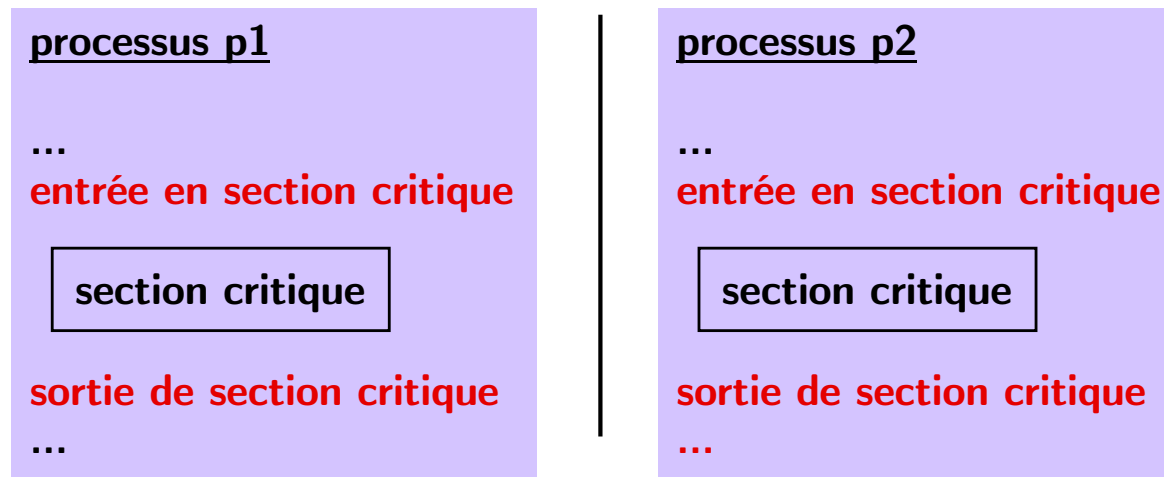
Si A1 et A2 sont atomiques, le résultat de l'exécution parallèle de A1 et A2 ne peut être que celui de A1 ; A2 ou de A2 ; A1, à l'exclusion de tout autre.

On dit aussi que la séquence d'actions 1; 2; 3 (dans p1 et dans p2) est une **section critique** : elle doit être exécutée en **exclusion mutuelle** (un seul processus au plus peut être dans sa section critique à un instant donné).

Réalisation d'une section critique (1)

► Schéma général

déclaration et initialisation de variables communes



Les opérations "*entrée en section critique*", "*sortie de section critique*" doivent garantir l'exclusion mutuelle

Réalisation d'une section critique (2)

Il existe plusieurs modes de réalisation d'une section critique

- par **attente active** :
 - Un processus qui attend l'autorisation d'entrée en section critique boucle sur le test d'entrée
 - Méthode **très inefficace** s'il y a un seul processeur
 - Méthode utilisée en multiprocesseur dans certains cas (pour des sections critiques très brèves)
- par **attente passive** :
 - Un processus qui attend l'autorisation d'entrée en section critique est bloqué jusqu'à ce qu'il puisse être admis en section critique (ou, au minimum, jusqu'à ce qu'il ait une chance d'être admis en section critique)
 - Méthode la plus efficace dans la plupart des cas

Un système d'exploitation fournit des primitives de base pour la gestion d'une section critique. La plupart de ces primitives sont spécifiques à une approche donnée (attente active ou attente passive). Certaines primitives sont compatibles avec les deux approches.

Réalisation d'une section critique (3)

Un système d'exploitation fournit des primitives de base pour la gestion d'une section critique.

Il existe différents types de primitives :

- Primitives de synchronisation générales :
 - Opérations sur **verrous**, **sémaphores**, **variables de conditions** (seront vues en M1)
- Primitives de synchronisation spécialisées :
 - Exemple : **verrouillage de fichiers**, vu plus loin
- Autres primitives (non-spécifiques à un rôle de synchronisation)
 - Mais dont les propriétés peuvent être exploitées pour implémenter des techniques de synchronisation *ad hoc* (exemple à venir)

Il reste à garantir que les primitives sont elles-mêmes **atomiques**, par des mécanismes internes au noyau du système (masquage des interruptions, *Test&Set* en multiprocesseur, etc.) – Détails vus en M1

Réalisation d'une section critique (4)

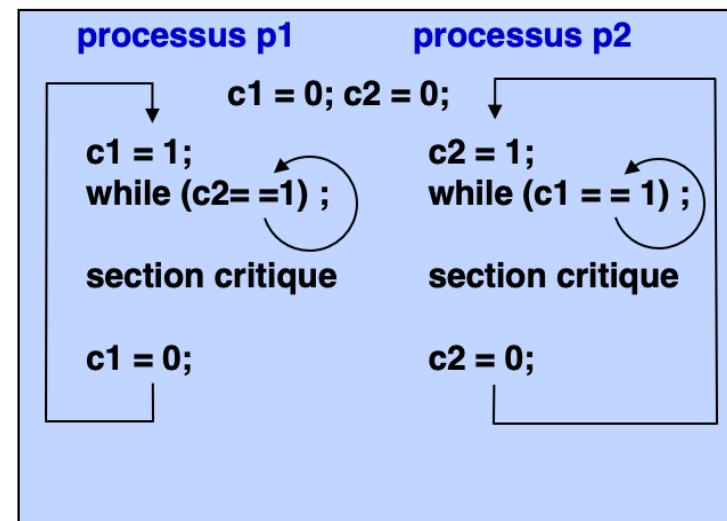
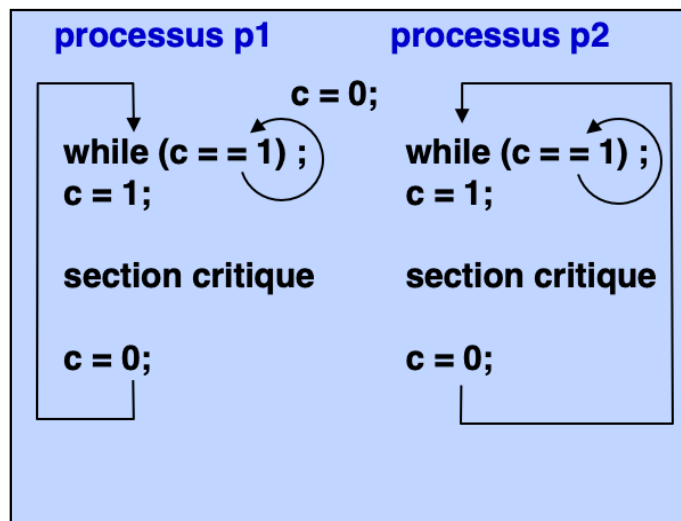
- ▶ un exemple (simplifié) de réalisation *ad hoc*:
 - ▶ Comment assurer qu'un seul navigateur Firefox est actif

```
/* lancer une session Firefox */  
if ((lock_descr = creat("~/mozilla/firefox/lock", 0)) == -1) {  
    /* afficher message d'erreur */  
    ...  
} else {  
    /* lancer navigateur */  
}  
...  
/* terminer session */  
close (lock_descr); unlink ("~/mozilla/firefox/lock");
```

- ◆ cet exemple utilise la propriété suivante : la primitive `creat` (création de fichier) est atomique (atomicité assurée par le noyau du système)

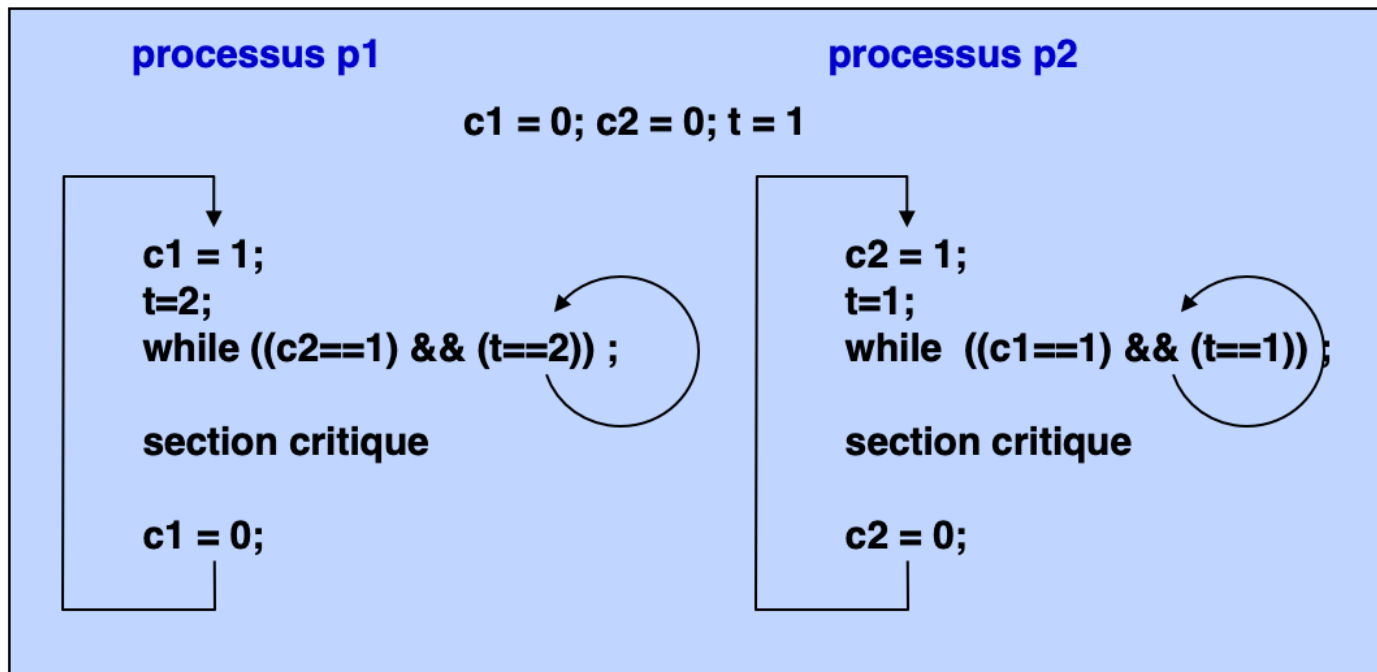
Réalisation d'une section critique par attente active (1)

- ▶ Réaliser l'exclusion mutuelle par attente active est plus difficile qu'il n'y paraît ...
- ▶ Exemples de “fausses solutions”, pour 2 processus



Réalisation d'une section critique par attente active (2)

- Une solution correcte pour l'exclusion mutuelle par attente active pour 2 processus (Peterson, 1981)



Verrouillage de fichiers

- ▶ Les opérations de verrouillage sont une manière de réaliser l'exclusion mutuelle, pour une opération particulière (l'accès à un fichier)
- ▶ Deux opérations
 - ▶ v-excl (f) : verrouille le fichier f avec accès exclusif
 - ▶ dev (f) : déverrouille le fichier f
 - ▶ Remarque : ces noms sont symboliques, voir plus loin réalisation en Unix
- ▶ **Propriétés garanties**
 - ▶ les opérations v-excl et dev sont **atomiques** (réalisées par appel système)
 - ▶ un fichier f verrouillé en accès exclusif par un processus ne peut pas être verrouillé par un autre processus
 - ▶ un processus qui tente de verrouiller un fichier déjà verrouillé en accès exclusif est bloqué (mis en attente du verrou)
 - ▶ l'opération de déverrouillage réveille un processus en attente du verrou

```
...
v-excl (f)
accès au fichier f (section critique)
dev (f)
```

... L3 SR

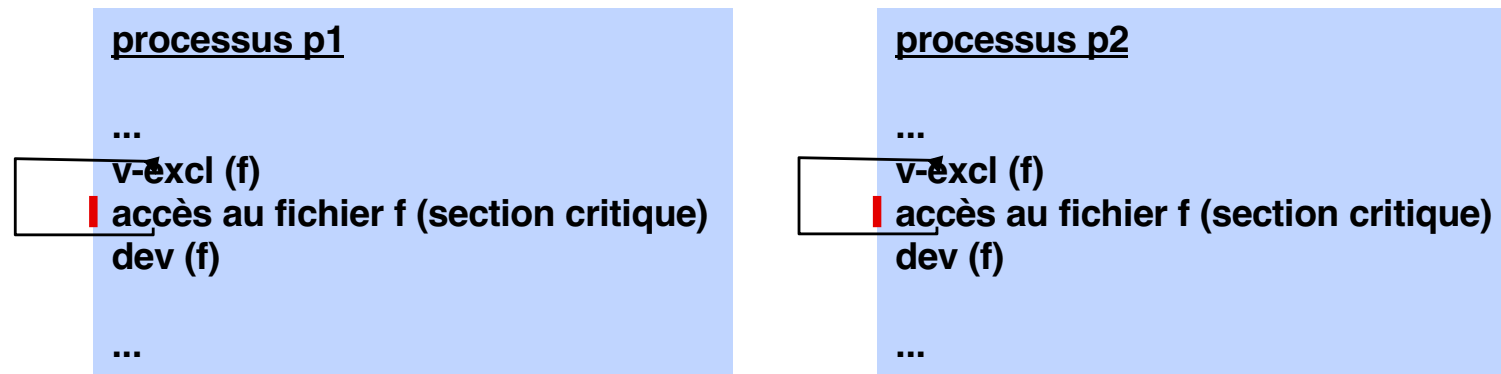
```
...
v-excl (f)
accès au fichier f (section critique)
dev (f)
```

...

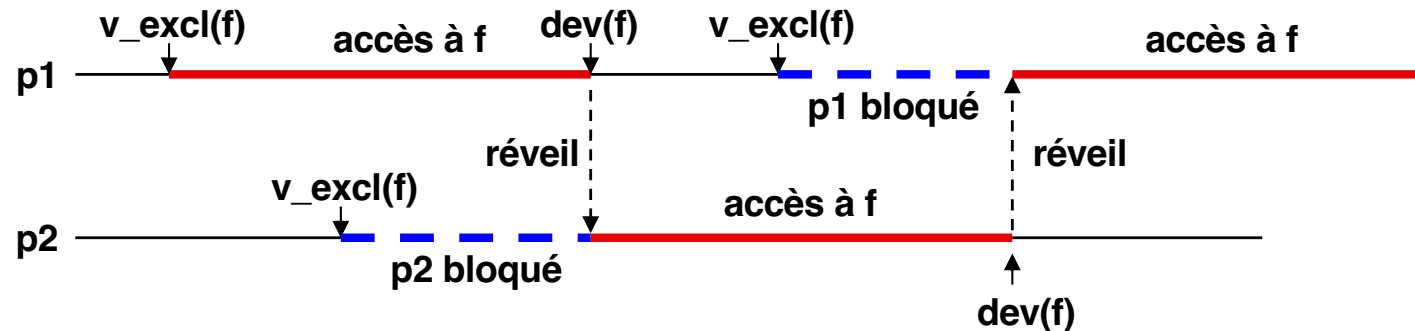
Verrouillage des fichiers (suite)

■ Rappel des opérations de verrouillage

les processus p1 et p2 partagent un fichier f, dans lequel ils écrivent
chaque séquence d'accès à f est une section critique (l'accès à f est
exclusif)



■ Fonctionnement



Verrouillage de fichiers dans Unix (simplifié)

► Opérations disponibles

- `fcntl` : primitive générale, complexe
- `lockf` : usage plus restreint (verrouillage exclusif seulement) ;
Caractéristiques

► Comment réaliser les opérations `v-excl(f)` et `dev(f)` ?

- Soit `fd` un descripteur de `f`, obtenu par `fd = open(f, mode)`, où `mode` indique le mode d'accès (détails dans cours sur systèmes de fichiers).
- Alors le verrouillage exclusif `v-excl` est obtenu par :

```
lockf (fd, F_LOCK, 0)
```

```
lockf (fd, F_TLOCK, 0)
```

- Différence entre `F_LOCK` et `F_TLOCK` : `F_LOCK` est bloquant si le fichier est déjà verrouillé ; `F_TLOCK` n'est pas bloquant et renvoie un code d'erreur

- Le déverrouillage `dev(f)` est obtenu par :

```
lockf (fd, F_ULOCK, 0)
```


Verrouillage de fichiers dans Unix : exemple

```
include <unistd.h>
#include <fcntl.h>
#define TRUE 1
int main(void) {
    int fd;
    fd = Open("toto", O_RDWR); /* doit exister */
    while(TRUE){
        if (lockf(fd, F_TLOCK, 0) == 0){
            printf("%d a verrouillé le fichier\n",
                getpid());
            sleep(5);
            if (lockf(fd, F_ULOCK, 0) == 0)
                printf("%d a déverrouillé le
                fichier\n", getpid());
            return;
        } else {
            printf("%d a trouvé le fichier déjà
            verrouillé, réessaie...\n", getpid());
            sleep (2);
        }
    }
}
```

testlock.c

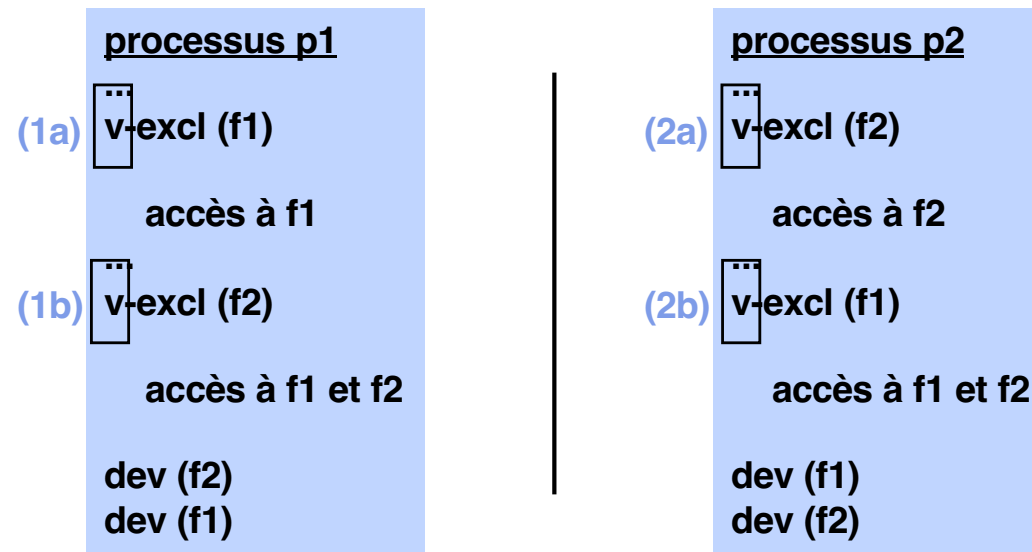
Question : que se passe-t-il
si on remplace F_TLOCK
par F_LOCK ?

```
<unix> testlock & testlock &
15545 a verrouillé le fichier
[4] 15545
15546 a trouvé le fichier déjà
verrouillé, réessaie
[5] 15546
<unix> 15546 a trouvé le fichier
déjà verrouillé, réessaie
15546 a trouvé le fichier déjà
verrouillé, réessaie
15545 a déverrouillé le fichier
15546 a verrouillé le fichier
15546 a déverrouillé le fichier
<unix>
```

Utilisation simultanée de plusieurs fichiers (1)

► Situation

- Les processus p1 et p2 partagent deux fichiers f1 et f2, et les utilisent en accès exclusif



■ Déroulement

- ◆ p1 et p2 s'exécutent en parallèle (ou pseudo-parallèle)

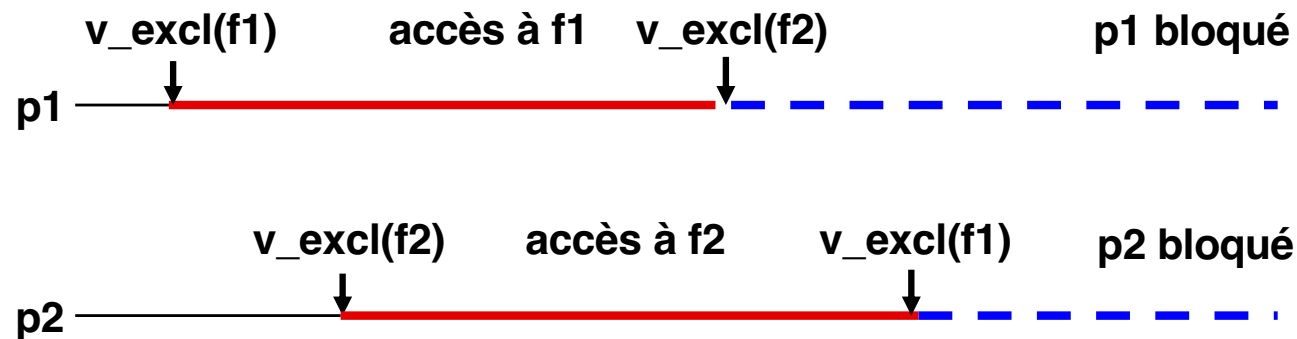
Pour une exécution particulière, la séquence est : 1a ; 1b ; 2a ; 2b ; ...

Pour une autre exécution, la séquence est : 1a ; 2a ; 1b ; 2b ; ...

Que se passe-t-il dans chaque cas ?

Utilisation simultanée de plusieurs fichiers (2)

► Exécution dans l'ordre 1a ; 2a ; 1b ; 2b ; ...



Situation après le deuxième **v_excl(f1)** : les deux processus p1 et p2 sont bloqués, et le resteront **indéfiniment** :

pour réveiller p1, il faut faire dev(f2) qui ne peut être fait que par p2, bloqué
pour réveiller p2, il faut faire dev(f1) qui ne peut être fait que par p1, bloqué

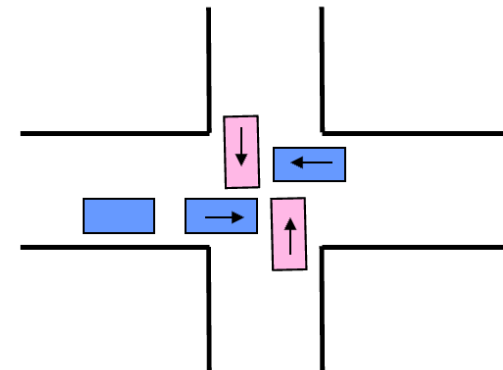
Cette situation s'appelle **interblocage** (en anglais *deadlock*)

Interblocage : caractérisation

- ▶ **Définition de l'interblocage**
 - ▶ Situation dans laquelle plusieurs processus (au moins 2) sont bloqués, chacun d'eux ne pouvant être réveillé que par une action de l'un des autres
 - ▶ On ne peut pas sortir d'une situation d'interblocage sans intervention extérieure
- ▶ **Conditions d'apparition**
 - ▶ L'interblocage se produit lorsque plusieurs processus sont en compétition pour utiliser simultanément plusieurs ressources, et que les demandes sont mal coordonnées (attente circulaire)
 - ▶ Un exemple hors informatique : carrefour

Questions :

que sont ici les ressources ?
comment sortir de l'interblocage ?



Lutter contre l'interblocage

- ▶ **On ne peut pas sortir d'une situation d'interblocage sans perdre quelque chose**
- ▶ **Possibilités de “guérison”**
 - ▶ Faire revenir un ou plusieurs processus en arrière, dans un état antérieur (on perd le travail réalisé depuis cet état)
 - nécessite d'avoir conservé l'état antérieur
 - ▶ Tuer un ou plusieurs processus pour libérer les ressources
- ▶ **Conclusion**
 - ▶ Pour choisir entre prévention et guérison, il faut apprécier les coûts relatifs de l'une et de l'autre dans la situation particulière
 - coût de la prévention : application d'une politique systématique de réservation de ressources
 - coût de la guérison : détection de l'interblocage + pertes résultant du retour en arrière

Introduction à la coopération entre processus

- ▶ Jusqu'ici, on n'a vu que des situations de **compétition** entre processus
 - ▶ pour l'utilisation du processeur
 - ▶ pour l'accès à un fichier
- ▶ Un exemple de **coopération** – le schéma producteur – consommateur



- ▶ **Utilité** : On cherche à augmenter le parallélisme entre producteur et consommateur, malgré une différence possible de leurs vitesses d'exécution
- ▶ **Fonctionnement**
 - Condition de blocage du consommateur : le tampon est vide
 - Condition de blocage du producteur : le tampon est plein (on ne peut pas y déposer de l'information sans "écraser" de l'information encore non consommée)
 - En fait, le fonctionnement est symétrique (le consommateur est un producteur de cases vides)
- ▶ **Applications**
 - Entrées-sorties tamponnées (spool)
 - Traitements en pipe-line : cf les tubes (pipes) d'Unix, à voir plus loin

Résumé de la séance 3

- ▶ **Réalisation de l'exclusion mutuelle par verrouillage**
 - ▶ accès à des informations en mode exclusif

- ▶ **Les problèmes de l'exclusion mutuelle**
 - ▶ Problèmes de réalisation (atomicité)
 - ▶ Problèmes d'utilisation (interblocage)

- ▶ **Introduction à la coopération entre processus**
 - ▶ schéma producteur-consommateur