Processus (suite)

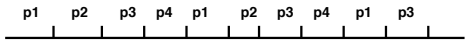
Problèmes élémentaires de synchronisation

Olivier Dalle
Université de Nice - Sophia Antipolis
http://deptinfo.unice.fr/
D'après le cours original de
Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/~krakowia

Raisonner sur les processus parallèles (important !)

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 (ou, ce qui revient au même, sur les vitesses relatives). Seuls comptent

- l'ordre d'exécution interne à chaque processus
- les relations logiques entre les processus (synchronisation)

Rappel : la synchronisation est réalisée en faisant attendre un processus (attente active, ou, mieux, blocage)

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

- **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

A1
```
1. courant = lire_compte (1867A)
2. nouveau = courant + 1000
3. ecrire_compte (1867A, nouveau)
```

processus p2

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

processus p1	processus p2
...	...
entrée en section critique	entrée en section critique
section critique	section critique
sortie de section critique	sortie de section critique
...	...

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** (le processus qui attend la section critique boucle sur le test d'entrée)
  - méthode très inefficace s'il y a un seul processeur
  - utilisé en multiprocesseur pour des sections critiques brèves, uniquement dans le noyau du système d'exploitation
- en utilisant des **primitives spéciales** fournies par le système, elles-mêmes atomiques
  - primitives générales : opérations sur sémaphores (sera vu en L3)
  - mécanismes adaptés à des situations spécifiques

    Exemple : verrouillage de fichiers, vu plus loin
  - il reste à garantir que les primitives sont elles-même atomiques, par des mécanismes internes au noyau du système (masquage des interruptions, Test&Set en multiprocesseur, etc.) - vu en L3

Réalisation d'une section critique (3)

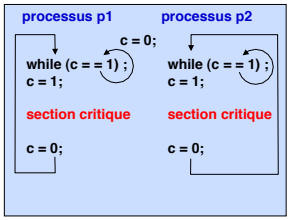
- **Il existe plusieurs modes de réalisation d'une section critique (suite)**
  - un exemple de réalisation "ad hoc" : comment assurer qu'un seul navigateur Mozilla/Firefox est actif

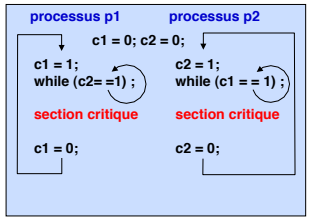
```
# lancer une session Firefox
try:
   lock_descr = os.open("~/.firefox/lock",
                os.O_RDWR|os.O_CREAT|os.O_EXCL)
   # succes : lancer navigateur...
except:
   # echec : message d'erreur
...
# fin session
os.close (lock_descr);
os.unlink ("~/.firefox/lock")
```

  - cet exemple utilise la propriété suivante : la creation `O_CREAT|O_EXCL` (création de fichier) est atomique (atomicité assurée par le noyau du système)

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

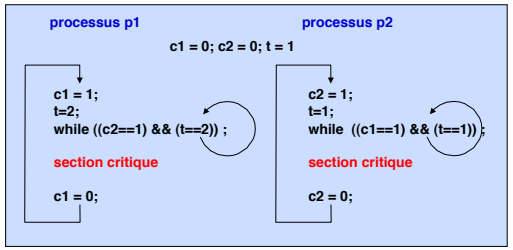
- **Réaliser l'exclusion mutuelle par attente active est plus difficile qu'il n'y paraît …**
- **Exemples de "fausses solutions", pour deux processus avec variables partagées (c, c1, c2)**





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

- **Une solution correcte pour l'exclusion mutuelle par attente active pour deux processus avec variables partagées c1, c2, t (Peterson, 1981)**



Opérations de verrouillage

- **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 leur 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 (et un seul)

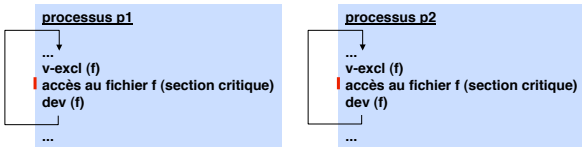
```
...
v-excl (f)
accès au fichier f (section critique)
dev (f)
...
```

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

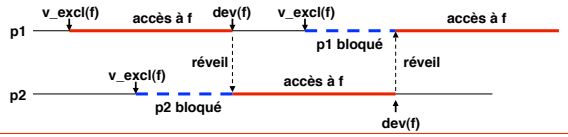

Verrouillage des fichiers (suite)

- **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)

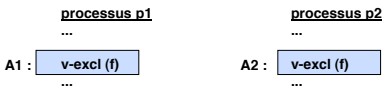


- **Fonctionnement**



Problèmes de réalisation du verrouillage

- **Les opérations `v-excl` et `dev` sont réalisées comme des opérations atomiques**
  - en particulier, si on exécute :



alors l'effet de l'exécution parallèle de A1 et A2 sera globalement : soit A1 ; A2, soit A2 ; A1, à l'exclusion de tout autre.

Cela est nécessaire au bon fonctionnement du mécanisme.

Exercice : essayer de réaliser les opérations `v-excl` et `dev`, en utilisant un indicateur d'occupation

`occ = 0` : l'accès est libre ; `occ = 1` : le fichier est occupé, accès impossible ; initialement, `occ = 0`

Quels sont les problèmes si on n'assure pas l'exécution atomique ?

nous avons déja rencontré une situation analogue

L'atomicité de `v-excl` et `dev` est assurée par le système d'exploitation (mécanisme interne aux "appels systèmes")

Verrouillage partagé

- **Le verrouillage exclusif n'est pas toujours nécessaire**
  - Supposons que p1 et p2 lisent le fichier `f` (sans le modifier), et que p3 écrive dans le fichier.
  - Alors on peut permettre l'accès simultané à `f` de p1 et p2 (mais non p1 et p3, ou p2 et p3)
- **Un nouveau mode de verrouillage : le verrouillage partagé : `v-part`**
  - un fichier `f` verrouillé en accès exclusif par un processus ne peut pas être verrouillé par un autre processus (en mode exclusif ou partagé)
  - un fichier `f` verrouillé en accès partagé par un processus ne peut pas être verrouillé par un autre processus en mode exclusif, mais peut être verrouillé en mode partagé
  - une opération de verrouillage bloque le processus qui l'exécute si l'une des règles ci-dessus s'applique
  - l'opération de déverrouillage réveille un processus en attente du verrou (et un seul)

Verrouillage de fichiers dans Unix

- **Opérations disponibles**
  - Deux types de verrous : exclusifs ou partagés
  - Deux modes de verrouillage
    - mode impératif (*mandatory*) : bloque les lectures ou écritures incompatibles avec les verrous présents ; c'est le mode qui a été décrit précédemment
    - mode consultatif (*advisory*) : ne bloque pas les lectures ou écritures (empêche seulement la pose de verrous incompatibles)
  - Deux primitives utilisables (voir `man`)
    - `fcntl` : primitive générale, complexe
    - `lockf` : usage plus restreint (verrouillage exclusif seulement) ; réalisé comme enveloppe de `fcntl`
- **Caractéristiques**
  - On peut verrouiller un fichier entier ou seulement une partie (permet d'augmenter le parallélisme)
  - Les verrous sur un même fichiers sont tous dans le même mode (impératif ou consultatifs)

Verrouillage de fichiers dans Unix : `lockf`

```
import fcntl
fcntl.lockf(fd, function, [length, [start, [whence]]])
```

`fd` : decripteur du fichier (ouvert) à verrouiller
`function` : mode de verrouillage
`fcntl.LOCK_UN` : déverrouiller
`fcntl.LOCK_EX` : verrouillage exclusif (impératif)
`fcntl.LOCK_SH` : verrouillage partagé
Combinable (OU binaire) avec `fcntl.LOCK_NB` : test seulement
`length` : nombre d'octets à verrouiller depuis la position start (si 0, tout le fichier, seule option que nous utiliserons)
`start` : indique à partir de quelle position vérouiller
`whence` : indique si start est compté depuis la fin, le début ou la position courante

En cas d'echec, `lockf` lève l'exception `IOError`.

Verrouillage de fichiers dans Unix : `lockf`

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, LOCK_EX, 0)
ou    lockf(fd, LOCK_EX|LOCK_NB, 0)
```

Différence: avec `LOCK_NB` le processus n'est pas bloqué (lève une exception `IOError`)

Le déverrouillage dev(f) est obtenu par :

```
      lockf(fd, LOCK_UN, 0)
```

Verrouillage de fichiers dans Unix : exemple

testlock.py

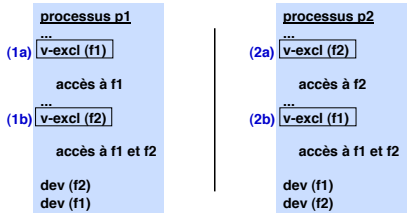
```
import os,sys,fcntl,time

fd = os.open("toto", O_RDWR);
while(True):
   try:
      fcntl.lockf(fd, fcntl.LOCK_EX|fcntl.LOCK_NB, 0)
      print "%d a verrouillé le fichier"%(os.getpid())
      time.sleep(5)
      fcntl.lockf(fd, fcntl.LOCK_UN, 0)
      print "%d a déverrouillé le fichier"%(ge
   except IOError:
      print "%d a trouvé le fichier déjà
         verrouillé, réessaie..."%(getpid())
      time.sleep (2);
```

Question : que se passe-t-il si on enlève `LOCK_NB` ?

```
<unix> ./testlock.py & testlock.py &
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



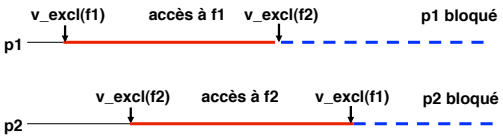
- **Déroulement**
  - p1 et p2 s'exécutent en parallèle (ou pseudo-parallèle)

    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 :

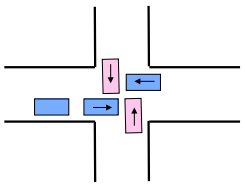
pour réveiller p1, il faut faire `dev(f2)` qui ne peut être fait que par p2, bloqué
pour réveiller p2, il faut faire `dev(f1)` qui ne peut être fait que par p1, bloqué

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

Interblocage : caractérisation

- **Définition de l'interblocage**
  - Situation dans laquelle plusieurs processus (au moins deux) 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 ?



Comment prévenir l'interblocage ?

- **Solution 1 : réservation globale**
  - Chaque processus demande globalement (en bloc) toutes les ressources dont il a besoin
  - Inconvénient : réduit les possibilités de parallélisme
  - Analogie du carrefour : *les feux*
- **Solution 2 : requêtes ordonnées**
  - L'interblocage est impossible si tous les processus demandent dans le même ordre les ressources dont ils ont besoin

demande f1…

… puis f2

processus p1	processus p2
...	...
v-excl (f1)	v-excl (f1)
accès à f1	accès à f1
...	...
v-excl (f2)	v-excl (f2)
accès à f1 et f2	accès à f1 et f2
dev (f2)	dev (f1)
...	...
dev (f1)	dev (f2)

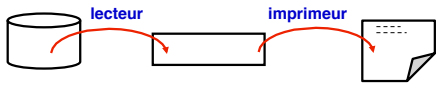
Analogie du carrefour : *le rond-point giratoire*

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

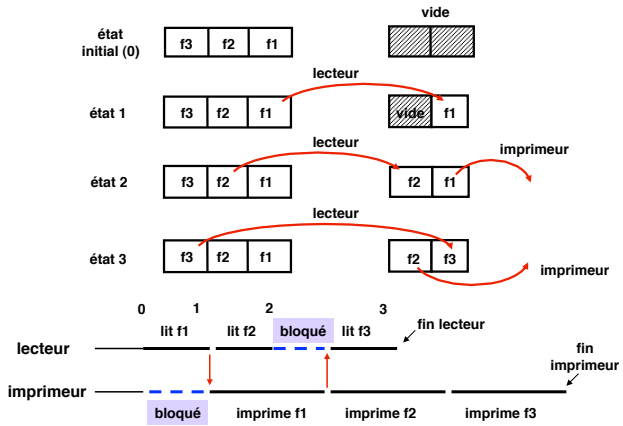
- **Jusqu'ici, on n'a vu que des situations de compétition entre processus**
  - pour l'utilisation du processeur
  - pour l'accès à un fichier
- **Un exemple de situation de coopération**
  - un processus *lecteur* lit un fichier depuis le disque vers la mémoire centrale
  - un processus *imprimeur* récupère le fichier dans la mémoire centrale et l'envoie à l'imprimante



- **Contraintes**
  - La place en mémoire est restreinte…
  - …mais on veut exécuter lecteur et imprimeur en parallèle

  - Solution : un "tampon" de n cases (ici n=2), et le fichier a une taille de 3 cases

Coopération entre processus



Le schéma producteur - consommateur

Le schéma lecteur -> tampon -> imprimeur est un exemple du schéma général de coopération dit "producteur - consommateur"

Utilité

- On cherche à augmenter le parallélisme entre producteur et consommateur, malgré une différence possible de leurs vitesses d'exécution
- La taille du tampon est d'autant plus grande que la différence de vitesses est grande

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**
  - **accès à des informations en mode exclusif ou partagé**
- **Les problèmes de l'exclusion mutuelle**
  - **Problèmes de réalisation (atomicité)**
  - **Problèmes d'utilisation (interblocage)**
- **Introduction à la coopération entre processus**
  - **schéma producteur-consommateur**