

LST Info&Miage

## 2<sup>ème</sup> partie en parallèle : Programmation système et réseau du point de vue « Multi- processus »

Françoise Baude

Université de Nice Sophia-Antipolis  
UFR Sciences - Département Informatique  
baude@unice.fr  
web du cours : deptinfo.unice.fr/~baude/SYSL3  
Février 2008

## Plan intervention Cours FB

| Séance de cours | Supports de cours  |
|-----------------|--|
| 4               | <a href="#">Introduction à la concurrence entre processus, et exclusion mutuelle</a>     |
| 7               | <a href="#">Introduction à la synchronisation, et utilisation de tubes</a>               |
| 8               | <a href="#">Difficultés de la concurrence : synchronisations complexes, interblocage</a> |
| 9               | <a href="#">Outils de synchronisation et API sous Unix</a>                               |
| 10              | <a href="#">Gestion de la mémoire virtuelle</a>  |
| 12              | <a href="#">Suite réseaux, partie "Couches basses"</a>                                   |
| 14              | <a href="#">Ordonnancement du CPU</a>  |
| 16              | <a href="#">Programmation multi-thread sous Unix, et lien avec multi-thread Java</a>     |

## Articulation p/r Cours PL

Point de vue relativement concentré sur les Interactions entre UN processus et le Système d'exploitation

|       |  |
|-------|--|
| 1     | <a href="#">Présentation du cours et chapitre 1 (introduction)</a> |
| 1 + 2 | <a href="#">Chapitre 2 (entrées-sorties)</a>                       |
| 2 + 3 | <a href="#">Chapitre 3 (système de fichiers)</a>                   |
| 3 + 5 | <a href="#">Chapitre 4 (Processus)</a>                             |

Point de vue concentré sur les Interactions entre PLUSIEURS processus, VIA le Système d'exploitation

|   |  |
|---|--|
| 4 | <a href="#">Introduction à la concurrence entre processus, et exclusion mutuelle</a>     |
| 6 | <a href="#">Chapitre 5 (Signaux)</a>   |
| 7 | <a href="#">Introduction à la synchronisation, et utilisation de tubes</a>               |
| 8 | <a href="#">Difficultés de la concurrence : synchronisations complexes, interblocage</a> |
| 9 | <a href="#">Outils de synchronisation et API sous Unix</a>                               |

## Introduction à la Concurrency entre processus & Exclusion Mutuelle

1. Introduction
2. Solutions avec attente active, dites Sans Arbitrage
3. Solutions avec blocage (attente passive), dites Avec Arbitrage

### 1 Introduction

## Exposé du problème

- Les entités (processus ou threads) en cours d'exécution sont généralement :
  - Indépendantes et Asynchrones
    - Leur fonctionnement ne dépend pas *a priori* du travail réalisé par les autres entités
    - Elles peuvent *a priori* progresser à leur rythme sans se soucier les unes des autres : elles pourraient s'exécuter en parallèle, même si sur un seul CPU, on parlera de « pseudo » parallélisme
- Pourtant, ces entités peuvent être en **concurrence** pour l'utilisation de **ressources**
- ⌘ Avoir besoin de se synchroniser et communiquer (chapitres suivants) : dans ce cas, elles seront au contraire dépendantes les unes des autres

### 1 Introduction

## La notion de « Concurrency »

- Définition :
  - « Rivalité d'intérêt entre entités provoquant une compétition »
  - Ressources physiques [à partager car en nombre insuffisant]
    - Processeur, disque, imprimante, ...
  - Ressources logiques [Leur rôle est de conserver ou mémoriser des données, un état commun]
    - Données globales du Système d'Exploitation
    - Zones de mémoire partagée
    - Fichiers d'une base de données

1 Introduction

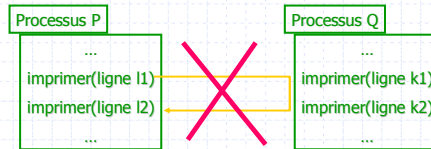
## Rôle des solutions proposées

- But: Contrôler la concurrence
  - Organiser la compétition:
    - Fournir des services de synchronisation indirecte par exclusion mutuelle : « Arbitrage », rôle du système
    - Ou au contraire, inclure la partie contrôle de concurrence au sein des programmes : « sans arbitrage » par le système
  - Coordonner l'utilisation des ressources:
    - Empêcher ou réparer des blocages, garantir l'équité ou absence de famine

1 Introduction

## Illustrations (1/3)

- Exemple 1 :
  - Pour tout  $i$ ,  $P_i$  ne doit pas imprimer tant que une autre impression est en cours !

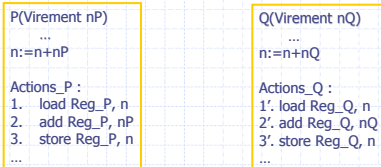


1 Introduction

## Illustrations (2/3)

- Exemple 2 :
  - Une instruction assembleur est indivisible, mais pas une suite d'instructions

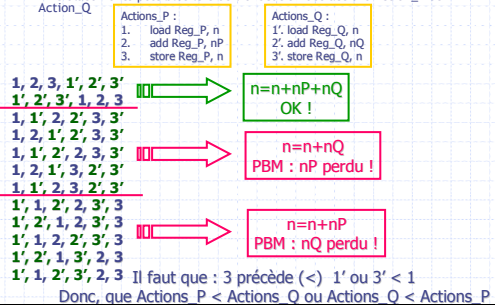
variable commune :  $n$  (compte bancaire)



1 Introduction

## Illustrations (3/3)

- Exemple 2 (suite)
  - Entrelacements possibles lors de l'exécution des actions Action\_P et Action\_Q



1 Introduction

## Solution au Problème

- Imposer que les sections critiques soient exécutées de manière non entrelacée
- pour garantir une utilisation correcte des ressources
  - L'imprimante dans l'exemple 1
  - La variable  $n$  dans l'exemple 2
- On dit aussi qu'une section critique doit être indivisible, atomique
  - Mais c'est un « abus de langage », car on ne veut pas forcément exécuter une section critique comme si c'était une (méga)-instruction => mauvaises performances (voir chapitre Ordonnement)

1 Introduction

## « Section Critique » et « Exclusion Mutuelle »

- Section (de code) Critique : code devant pouvoir faire l'hypothèse qu'il utilise la ressource de manière exclusive
  - Si aucune précaution particulière n'est prise, rien n'empêche plusieurs entités à la fois d'utiliser la ressource
    - ⇒ Empêcher les entités qui sont en compétition (pour une ressource donnée) d'entrer simultanément dans leur section [de code] critique.
    - ⇒ Les sections critiques s'exécutent donc en exclusion mutuelle

1 Introduction

## Classes de solutions au problème de l'Exclusion Mutuelle

- 2 grandes classes de solutions envisageables :
  - Solution Ad hoc, sans arbitrage, intégrée aux codes programmeur
  - Allocation de ressource incorporant appel implicite à un arbitre
- Les 2 classes exigent l'ajout de *code de protection* autour de la section critique:

|   |   |
|---|---|
| <b>Protocole d'entrée en section critique</b><br><section de code critique><br><b>Protocole de sortie de section critique</b> | <b>Demande de la ressource</b><br><section de code critique><br><b>Libération de la ressource</b> |
|---|---|

Remarque : dans certains cas (si ressource=CPU) la demande d'allocation de la ressource n'est pas explicite...

1 Introduction

## Propriétés attendues d'une solution (1/2)

- Exclusion mutuelle**  
A tout instant, un processus au plus exécute des instructions de sa section critique
- Absence de blocage (permanent)**  
Si plusieurs processus attendent pour entrer en SC, et si aucun processus n'est déjà en SC, alors un des processus qui attend doit pouvoir entrer en SC au bout d'un temps fini
- Condition de progression (cad. blocage que temporaire)**  
Un processus qui se trouve hors de sa SC et hors du protocole contrôlant l'accès à la SC ne doit pas empêcher un autre processus d'entrer dans sa SC : un processus ne doit pas ralentir un autre

1. Introduction

## Propriétés attendues d'une solution (2/2)

- Équité (Absence de famine)**  
Un processus qui est bloqué à l'entrée de la section critique n'attendra pas indéfiniment son tour  
  
Pour un processus qui veut entrer en SC, il existe une borne supérieure au nombre de fois où d'autres entités exécuteront leur SC avant lui  
(La valeur de la borne permet de mesurer à quel point une solution est équitable)
- Code identique (propriété souhaitable par souci de simplicité)**  
Le code qui protège la section critique (protocole d'entrée et protocole de sortie) est le même pour toutes les entités

## Introduction à la Concurrency entre processus & Exclusion Mutuelle

- Introduction
- Solutions avec attente active, dites Sans Arbitrage
- Solutions avec blocage (attente passive), dites Avec Arbitrage

2 Solutions avec attente active

## Protocoles de gestion de Section Critique

- Principe :  
Une entité désirant entrer en SC attend de façon active qu'une condition soit vérifiée (ici, aucune autre entité en SC)

|   |
|---|
| <b>Tant que</b> (condition indique SC non libre) <b>Faire</b><br>rien ou sleep(court délai)<br><b>Fintantque</b><br><section de code critique><br>Modifier condition pour refléter SC libre<br><section de code non critique> |
|---|

Problème : Consommation inutile de temps CPU

2 Solutions avec attente active

## Exemple simple et incorrect...

Libre: variable commune, initialement Vraie

Incorrect: P1 & P2 peuvent continuer dans leur SC en "mauvais" temps

2 Solutions avec attente active

### Solutions dites « logicielles » (1)

- Algorithme 1 : une solution « naïve »

Tour : variable commune, initialement =1 (par ex)  
(quand tour =i, Pi peut entrer en SC)

**P1**

**Répéter**  
**Tant que** (Tour = 2) **Faire**  
sleep(délai);  
**Fintantque**  
<section critique>  
Tour :=2  
**Jusqu'à** faux

**P2**

**Répéter**  
**Tant que** (Tour = 1) **Faire**  
sleep(délai);  
**Fintantque**  
<section critique>  
Tour :=1  
**Jusqu'à** faux

2 Solutions avec attente active

### Solutions dites « logicielles » (2)

- Analyse algorithme 1
  - Exclusion mutuelle : Ok
  - Absence de blocage : Ok
  - Condition de progression : **NON**
    - Lorsque Pi quitte sa SC, Tour:=j
    - Pour entrer à nouveau en SC, Pi doit attendre que Pj exécute sa SC (stricte alternance des 2)
  - Absence de famine : Ok
  - Code identique : **NON**

### Solutions dites « logicielles » (3) [Peterson 1981]

- Correct, généralisable à N processus (mais N fixé à l'avance...)

Tour : variable commune, initialement =1 (par ex)  
Drapeau1, Drapeau2, initialement = faux  
(quand tour =i, Pi peut entrer en SC)

**P1**

...  
**Drapeau1** = vrai ; Tour:=2  
**Tant que** (Drapeau2 ET Tour=2)  
**Faire** sleep(délai);  
**Fintantque**  
<section critique>  
Drapeau1 := faux  
...

**P2**

...  
**Drapeau2** = vrai ; Tour:=1  
**Tant que** (Drapeau1 ET Tour=1)  
**Faire** sleep(délai);  
**Fintantque**  
<section critique>  
Drapeau2 := faux  
...

2 Solutions avec attente active

### Solutions dites « logicielles » (4)

- Analyse algorithme de Peterson
  - Exclusion mutuelle
  - preuve par l'absurde : P1 ET P2 en SC

P1 en SC =>  
Drapeau2=faux OU Tour =1

P2 en SC =>  
Drapeau1=faux OU Tour =2

P1 en SC =>  
Drapeau1 = vrai

P2 en SC =>  
Drapeau2 = vrai

Donc Tour = 1 ET Tour = 2 => Impossible !

2 Solutions avec attente active

### Solutions dites « logicielles » (5)

- Analyse algorithme Peterson (suite)
  - Absence de blocage
  - Preuve par l'absurde : supposons P1 et P2 sont incapables de terminer leur protocole d'entrée en SC (cad bloqués simultanément) dans Tant que

Drapeau2=vrai  
ET Tour = 2

ET

Drapeau1= vrai  
ET Tour = 1

Donc Tour = 1 ET Tour = 2 => Impossible !

2 Solutions avec attente active

### Solutions dites « logicielles » (6)

- Analyse algorithme Peterson (suite)
  - Condition de progression
  - Preuve par l'absurde : supposons P2 dans sa section restante (hors SC) ET P1 incapable de finir son protocole d'entrée, cad :

Drapeau2 = faux

Drapeau2=vrai ET Tour = 2

Donc Drapeau2 = faux ET Drapeau2 = vrai

=> Impossible !

2 Solutions avec attente active

### Solutions dites « logicielles » (7)

- Analyse algorithme Peterson (suite)
  - Absence de famine

Preuve « directe » : Supposons que P2 attend alors que P1 est dans sa SC.

**Drapeau2 = vrai** ET **P2 attend : en effet**  
**Drapeau1= vrai ET Tour = 1**

Maintenant, P1 quitte sa SC, puis exécute à nouveau le protocole d'entrée en SC :

**Drapeau2=vrai ET Tour = 2**

P1 NE peut PAS entrer à nouveau en SC. P2 POURRA entrer dès son prochain tour de boucle (test de la condition).

2 Solutions avec attente active

### Solutions de niveau Matériel

- Masquage des niveaux d'interruption
  - Protocole d'entrée en SC : masquer les interruptions
  - La SC ne peut être interrompue car les changements de contexte sont impossibles
  - Protocole de sortie de SC : démasquer les interruptions
- Remarques :
  - Solution uniquement utilisable dans (par) l'exécutif
  - A utiliser pour des SC très courtes, car plus de parallélisme
  - Ne convient pas pour les machines multi-processeurs à mémoire partagée (car chaque processeur gère ses propres interruptions)
- Analyse:
  - Absence de blocage : OK, l'ordonnanceur du CPU finit par donner la main à un processus voulant rentrer dans sa SC
  - Condition de progression : OK
  - Absence de famine : OK si l'ordonnanceur l'assure (normalement oui !)
  - Code identique : OK

2 Solutions avec attente active

### Solutions de niveau Matériel (2)

- Utilisation d'une instruction CPU : *Test-And-Set (TAS)*
  - `prev = TAS(var)` exécuté de façon réellement indivisible :
    - `prev = var`
    - `var = 1`
  - Intérêt : deux (plusieurs) entités ne peuvent pas (avoir l'impression de) faire passer la variable de la valeur 0 à la valeur 1 en même temps :
    - Celui qui fait passer la variable de la valeur 0 à la valeur 1 le sait (en sortie, `prev = 0`)
    - Celui qui NE peut PAS faire passer la variable de la valeur 0 à la valeur 1 le sait aussi (en sortie, `prev = 1`)
- ⇒ Celui qui gagne est celui parvient à faire passer la valeur de la variable de 0 à 1 (celui qui récupère `prev = 0`).

2 Solutions avec attente active

### Solutions de niveau Matériel (3)

- Utilisation de Test-and-set en attente active pour l'exclusion mutuelle

verrou: variable commune, initialement 0 (« déverrouillé »)

|  |  |
|--|--|
| <p><b>P1</b></p> <pre>prev : var locale ... prev = TAS(verrou); <b>Tant que</b> (prev = 1) <b>Faire</b> /* sleep(délai); */ prev = TAS(verrou); <b>Fintantque</b> &lt;section critique&gt; verrou = 0; ...</pre> | <p><b>P2</b></p> <pre>prev : var locale ... prev = TAS(verrou); <b>Tant que</b> (prev = 1) <b>Faire</b> /* sleep(délai); */ prev = TAS(verrou); <b>Fintantque</b> &lt;section critique&gt; verrou = 0; ...</pre> |
|--|--|

2 Solutions avec attente active

### Solutions de niveau Matériel (4)

- Analyse de la solution basée sur TAS :
  - Exclusion mutuelle : Ok
  - Absence de blocage : Ok
  - Condition de progression : Ok
  - Absence de famine :
    - Ok si l'ordonnanceur l'assure (cad si n'active pas  $P_i$  éternellement),
    - sinon, c'est NON
      - si  $P_i$  quitte sa SC, garde le CPU et se présente à nouveau à l'entrée de la SC : verrou toujours à 0, donc  $P_i$  entre à nouveau... (remarque applicable aussi pour sol. « Masquage »)
  - Code identique : Ok

2 Solutions avec attente active

### Solutions de niveau Matériel (5)

- Remarques sur solution avec TAS :
  - Sur une machine multi-processeurs à mémoire partagée, plusieurs exécutions de TAS peuvent s'entrelacer
    - ⇒ L'Exclusion Mutuelle n'est plus garantie
    - Remède (matériel aussi) : verrouillage du bus mémoire pendant l'instruction TAS
  - Le mécanisme pour résoudre l'exclusion mutuelle (ici TAS) est lui-même une section critique, qui doit parfois être protégée à un niveau plus bas (ici verrouillage bus mémoire)
  - En fait, ce principe est généralisable pour toute solution au problème de l'exclusion mutuelle (voir TD)

## 2 Solutions avec attente active

### Bilan

- Solutions avec attente active gourmandes en temps CPU
- Solutions ad-hoc:
  - programmation plus technique, source d'erreurs si on oublie de libérer la SC
- L'équité dépend souvent entièrement de l'ordonnanceur, or il n'est pas explicitement sollicité
  - Un processus peut garder le CPU suffisamment longtemps et faire ainsi plusieurs entrées en SC sans que ceux qui attendent puissent avoir leur tour

### Bilan sur Exc. Mut par Attente active

- test « dois-je attendre ? » est exécuté en boucle
  - Ne provoque pas de commutation de contexte volontaire (« je rends la main ») de la part du processus qui attend
    - Sauf que si on veut qu'un processus fasse progresser les choses et donc que l'attente se termine, il faut bien qu'il ait la main, donc, que le SE commute vers lui !
    - Utilise / gaspille du temps CPU (jusqu'à épuisement du quantum de temps)
    - A utiliser de préférence pour des attentes qu'on sait être courtes car la condition va changer rapidement ...
      - Ex: ... si la section de code critique est toujours très courte
      - Ex: ... si l'événement qui va rendre la condition vraie arrive très fréquemment
    - et si ce changement peut arriver en parallèle de l'attente
      - Ex: cas d'une machine multi-CPU
      - Ex: événement provoqué par un autre élément que le CPU (ex, fin d'E/S disque)

## Introduction à la Concurrency entre processus & Exclusion Mutuelle

1. Introduction
2. Solutions avec attente active, dites Sans Arbitrage
3. Solutions avec blocage (attente passive), dites Avec Arbitrage

## 3 Solutions avec blocage Généralités

- Les solutions avec attente active sont gourmandes en temps CPU
  - Lorsque l'attente est longue
  - Lorsque plusieurs (nombreux) concurrents attendent
- Les processus épuisent leur quantum dans l'attente active

## 3 Solutions avec blocage

### Principe de fonctionnement

#### Répéter

- ...
  - ① Est-ce que je peux entrer en SC (je demande ceci à l'arbitre) ?
  - ② Si NON alors je me bloque ...  
<section critique>
- Je relâche la SC (en le disant à l'arbitre), et la question se pose :
  - Est-ce qu'un processus est bloqué ?
  - Si OUI alors le réveiller (ou Ordonner un réveil)
- Jusqu'à faux

- Attention au problème des « Ordres de Réveil Perdus » (lost wake-up) (revu en TD):
- Au moment d'exécuter ②, le résultat de ① n'est peut-être plus valide !!
  - La seule solution pour éviter le problème du réveil perdu :
    - Réaliser de manière indivisible le test permettant de décider du blocage et le blocage lui-même (cad le protocole d'entrée)

## 3 Solutions avec blocage

### Principe de fonctionnement (2)

- Le protocole de sortie doit aussi être réalisé de manière indivisible
  - Modification de l'état d'occupation de la SC et réveil d'un processus doivent se réaliser de manière indivisible
  - Sinon : risque d'avoir plus de 1 processus en SC !
- Plusieurs mises en œuvre de ce principe avec blocage ...
- ... Mais toujours en étroite relation avec le support d'exécution des entités
  - processus : Noyau du système
  - threads : Noyau librairie threads (ou Noyau du système si threads gérées par lui)

### 3 Solutions avec blocage

## Sémaphores [Dijkstra, 1965]

- Outil général, pouvant servir à réguler d'autres interactions de nature « Synchronisation » entre entités :
  - permettre à un nombre borné (éventuellement plus grand que 1) d'entités d'entrer en section critique
  - Attendre qu'un nombre minimal d'entités soient bloquées avant que l'une d'elles puisse continuer (rendez-vous)
  - ...
- Description
  - Sémaphore = type abstrait
    - structure de données (compteur + file d'attente d'entités)
    - interface (opérations, dont initialisation, sur la structure de données)

### 3 Solutions avec blocage

## Sémaphores (2)

- structure de données :
  - Val : entier **signé**, indique un nombre de permissions
  - File : file de (contexte ou descripteur) entités
- opérations :
  - init(s:sémaphore, val\_init : entier)
    - s.Val = val\_init
  - P(s: sémaphore) **indivisible**
    - « Prendre » le sémaphore
  - V(s: sémaphore) **indivisible**
    - « Relâcher » le sémaphore
- Remarque : P vient du hollandais *Proberen* (essayer) et V vient de *Verhogen* (libérer)

### 3 Solutions avec blocage

## Sémaphores: opération P

```
Opération P(s:sémaphore)  
Si (s.Val <= 0)  
Alors //ranger le descripteur du processus dans s.File  
ajouter (s.File, descripteur processus courant);  
//mettre le processus dans l'état bloqué  
descripteur courant . Etat = BLOQUÉ  
//choisir un nouveau processus à rendre actif  
faire appel à l'ordonnanceur pour qu'il choisisse  
FinSi  
FinSi  
s.Val := s.Val - 1;
```

### 3 Solutions avec blocage

## Sémaphores: opération V

```
Opération V(s:sémaphore)  
Si (s.File non vide)  
Alors //retirer de s.File le descripteur d'un processus  
P= retirer (s.File)  
// mettre ce processus dans l'état prêt  
P. etat = PRÊT  
FinSi  
s.Val := s.Val + 1;
```

### 3 Solutions avec blocage

## Sémaphores : utilisation pour résoudre l'exclusion mutuelle

- Exclusion mutuelle = 1 seule permission accordée
- Variable commune : **Mutex** : sémaphore
- Avant de commencer, une entité exécute **INIT(Mutex,1)**
- **Mutex.Val = 1** => SC libre
  - **Mutex.Val = 0** => SC occupée

Pi pour tout i :

```
Répéter  
<hors section critique...>  
P(Mutex)  
<section critique>  
V(Mutex)  
<hors section critique...>  
Jusqu'à ...
```

### 3 Solutions avec blocage

## Analyse solution par sémaphores

- Exclusion mutuelle : Ok par définition de P et de **INIT(Mutex,1)**
- Absence de blocage : Ok par indivisibilité de P
- Condition de progression : Ok par définition de V
- Absence de famine : dépend de l'ordre de retrait de la file du sémaphore. Si l'ordre est FIFO = Ok
- Code identique : Ok

Et c'est efficace ! Pendant l'attente des processus, ceux qui ne sont pas bloqués peuvent utiliser le CPU

### 3 Solutions avec blocage

## Moniteurs

[Hoare, 1974 - Brinch Hansen, 1975]

- **Motivation** : à cette époque, émergence du concept de type abstrait: structure de donnée cachée manipulée par des opérations de spécification connue (précurseur d'objet!)
- **D'où l'idée évidente** : intégrer les protocoles d'entrée et de sortie de SC au début et à la fin des opérations spécifiées comme devant s'exécuter en exclusion mutuelle
  - C'est le compilateur qui rajoute les instructions du protocole d'entrée et de sortie, souvent à base de sémaphores.
- Utilisation pour l'exclusion mutuelle : immédiat par définition !

### 3 Solutions avec blocage

## Moniteurs (2)

- Exemple de moniteur : gestion de compte bancaire

```
Type Compte : moniteur
variable solde : entier /* ressource à accès exclusif */

procédure Etat /* en exclusion mutuelle avec Virement */
début
  imprimer solde;
fin

procédure Virement(montant : entier) /* en exc. mut. avec Etat & Virement */
début
  solde := solde + montant;
fin

début
  solde := 0; /* Initialisation du moniteur */
fin
```

### 3 Solutions avec blocage

## Moniteurs (3)

- Exemple (suite)  
Activités concurrentes :

```
Activité P :
...
Un_compte.Virement(nP);
...
```

```
Activité Q :
...
Un_compte.Virement(nQ);
Un_compte. Etat();
...
```

```
Activité R :
...
Un_compte.Virement(nR);
...
Un_compte. Etat();
...
```