

# Problèmes classiques

## ASR7-Programmation concurrente

Matthieu Moy, Fabien Rico, Adil Khalfa

Univ. Claude Bernard Lyon 1

séance 2

Matthieu Moy	matthieu.moy@univ-lyon1.fr	CM + TD + TP
Fabien Rico	fabien.rico@univ-lyon1.fr	TD + TP
Adil Khalfa	adil.khalfa@cc.in2p3.fr	TP



## 1 Introduction

## 2 Problèmes

- Problèmes classiques

## 3 Critères à surveiller

- Famine (*starvation*)
- Interblocage (*deadlock*)



# Introduction

On étudie souvent un certain nombre de problèmes classiques de synchronisation

- Car beaucoup de problèmes réels s'en rapprochent.
- Car leurs solutions peuvent se comparer et se prouver :
  - ▶ absence d'interblocage ;
  - ▶ favoritisme ou non ;
  - ▶ coût pour les mettre en œuvre ou les utiliser.
  - ▶ ...



## 1 Introduction

## 2 Problèmes

- Problèmes classiques

## 3 Critères à surveiller

- Famine (*starvation*)
- Interblocage (*deadlock*)



# Cas d'école

- Problème de la barrière.
- Problème des philosophes (dîner ou buffet)
- Problème des producteurs et consommateurs.
- Problème des lecteurs et rédacteurs.
- ...



# Producteur consommateur

## Description

- Un ou des producteurs produisent des informations
- consommées par un ou des consommateurs.
- L'échange utilise une FIFO de taille limitée.
- Les consommateurs sont bloqués si la file est vide.
- Les producteurs se bloquent si la file est pleine.

Exemple : Gestion d'une file d'attente partagée, communication entre entité, organisation maître esclave ...



# Lecteurs et rédacteurs

## Description

- 2 types d'accès (lecture, écriture),
- 1 seul *type* d'accès à la fois,
- tous les lecteurs,
- ou, un seul rédacteur,
- exclusion entre les lecteurs et le rédacteur,
- exclusion entre les rédacteurs.

Exemple : une base de données, accès à des disques réseaux, gestion des caches ...



# Problème de la barrière

## Description

Plusieurs threads doivent se donner rendez-vous :

- un certain nombre de threads est attendu,
- tous les threads arrivent sur la barrière et se bloquent,
- le dernier arrivé libère tout le monde.

Exemple : traitements parallèles, où les threads les plus rapides doivent attendre les autres lors de certains *points de synchronisation*.





# Buffet des philosophes

## Description

- Un ensemble de philosophes pensent (mais sont affamés), mangent, ou préparent un plat.
- Pour manger ils utilisent des mets disposés sur un buffet (c'est la ressource partagée).
- Deux philosophes ne peuvent pas utiliser le même met en même temps.
- Comment assurer que chaque philosophe arrive à préparer ses plats en un temps fini ?

Exemple : gestion d'un pool de ressources (disques, carte réseau, processeurs, ...) par un pool de threads.



## 1 Introduction

## 2 Problèmes

- Problèmes classiques

## 3 Critères à surveiller

- Famine (*starvation*)
- Interblocage (*deadlock*)



# Synchronisation : application aux lecteurs et rédacteurs



# Synchronisation : application aux lecteurs et rédacteurs

## Exemple (équité ?)

Supposons un lecteur/rédacteur. La présence des lecteurs empêche les rédacteurs d'entrer en section critique.

- que se passe-t-il lorsque les lecteurs arrivent régulièrement,
- et qu'un rédacteur attend ?



## Famine/Favoritisme

L'utilisation de la synchronisation a pour but de bloquer des threads.  
⇒ il faut s'assurer que tout thread sera débloqué en temps *raisonnable*.

### Definition (Famine)

Il y a *famine* quand certains threads ou classes de threads ne peuvent pas obtenir l'accès à la section critique durant un temps impossible à borner.

Exemple : circulation avec priorité en cas d'affluence.

### Definition (Favoritisme)

Il y a *favoritisme* quand certaines classes de threads ont plus facilement accès à la section critique.

Exemple : lecteurs/rédacteur, panneau stop sur la route.



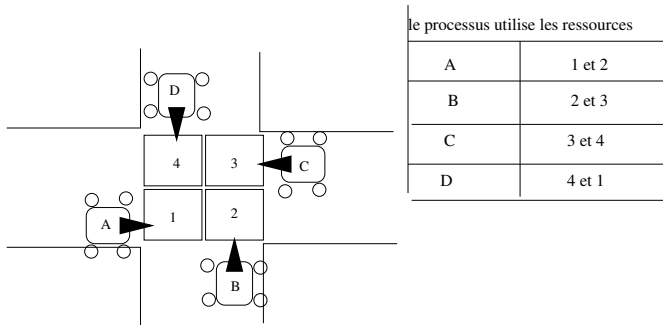
## Autre problème

### Exemple (Carrefour)

- Un carrefour avec 4 embranchements ;
- 4 voitures se présentent en même temps ...



## Le carrefour : la situation



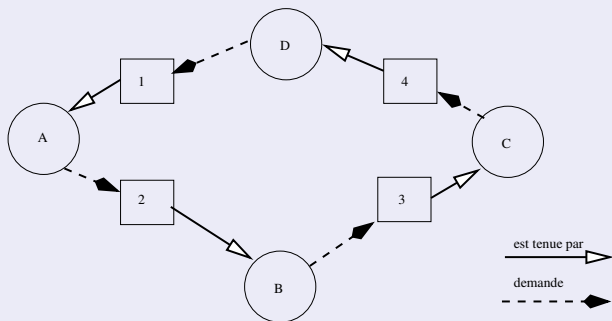
### analogie

- processus : voiture
- ressource : partie du carrefour (représentée ici par un verrou)
- définition d'une ressource : élément nécessaire à l'avancement d'un processus et pouvant provoquer une attente

# L'interblocage

## Définition et conditions d'existence

- *exclusion mutuelle* : un seul processus au plus possède la ressource.
- *Tenir et attendre* : un processus en attente ne relâche pas les ressources qu'il détient.
- *Non préemption* : une ressource attribuée n'est pas reprise.
- *Cycle d'attente* :



Attente circulaire



# L'interblocage : préventions

## Comment ?

supprimer une des conditions d'existence :

- Supprimer l'*Exclusion mutuelle* : généralement pas possible !
- Éviter le fait de *Tenir et attendre* :
  - ▶ Un processus qui ne peut pas obtenir toutes ses ressources nécessite libérer celle qu'il détient.
  - ▶ On oblige de prendre (ou relâcher) toutes les ressources nécessaires en 1 fois. La décision est centralisée dans un moniteur.

Mais ces deux solutions peuvent provoquer la famine des processus gourmands.

- Autorisez la *préemption* : utilisation d'une autorité de régulation (gendarme).
- prévenir l'apparition de cycles : par exemple ordonner les ressources et toujours les prendre par ordre croissant :
  - ▶ A prend 1 avant 2,
  - ▶ B prend 2 avant 3,
  - ▶ C prend 3 avant 4,
  - ▶ D prend 1 avant 4 (la plus lointaine d'abord).



# Interblocage

## Détection et suppression

- recherche de cycles dans le graphe d'allocation/demande
- que faire ensuite ? tuer un des processus ? lequel ? et il faudra le relancer
- plus finement : retour arrière d'un ou plusieurs processus (moniteur transactionnel de bases de données), mais lourd à mettre en oeuvre dans le cas général



# Interblocage

## Détection et évitement

- lors d'une requête, vérifier que celle-ci ne crée pas un cycle (mais que faire dans ce cas)
- Algorithme du Banquier (suppose qu'on connaît les demandes de tout le monde).



## Synchronisation : conclusion

### Il y a toujours de l'exclusion mutuelle

- la différence est la durée de l'accès exclusif :
- soit toute l'action est en accès exclusif (et ça, c'est mal),
- soit la prise de décision se fait en accès exclusif, mais pas l'action (c'est mieux !)
- la meilleure synchronisation est l'absence de synchronisation ! (mais ce n'est pas toujours possible ...)



# Synchronisation : objectifs

## Qualités d'une synchronisation / arbitrage

- pas de *famine* : tout le monde réalise ce qu'il a demandé ...
- équitable : ... sans favoritisme.
- avancement : un processus ne doit pas être mis en attente si ce n'est pas nécessaire ...

Parfois, le favoritisme est utile (exemple de l'annuaire).

De toute façon, la synchronisation est délicate à réaliser en pratique :

⇒ il faut utiliser une bibliothèque testée et approuvée :

```
// En C++
```

```
std::shared_mutex m;  
m.lock() / m.lock_shared()
```

```
// En C
```

```
pthread_rwlock_rdlock() / unlock()  
pthread_rwlock_wrlock() / unlock()
```



## 1 Introduction

## 2 Problèmes

- Problèmes classiques

## 3 Critères à surveiller

- Famine (*starvation*)
- Interblocage (*deadlock*)



# À retenir

## Thread

- Différences thread/processus
- Programmation multithread

## Synchronisation

- Section critique et exclusion mutuelle
- Utilisation des mutexes et Variables de condition
- Moniteur
- Problème des Producteurs/consommateurs, Lecteurs/rédacteurs...
- Interblocage

