Exclusion Mutuelle

Arnaud Labourel
Courriel : arnaud.labourel@lif.univ-mrs.fr

Université de Provence

9 février 2011

Contexte “Epistémologique”

Comment être sûr qu'un programme est **vraiment** correct ?

Sections Critique et Exclusion Mutuelle

Etant donné du code, on peut y définir des portions particulièrement importantes que l'on dénote par *section critique* (**SC**).

Exemple: manipulation d'une variable partagée.

```
/* Du code */
...

/* Section Critique */
... // variable partagée
/* Fin de la Section Critique */
/* Encore du Code */
...
```

Une solution

Exclusion Mutuelle : Il y a au plus une entité en section critique.

$\Rightarrow$ Permet de réguler l'accès aux données.

Mais ...

Cette spécification est incomplète pour pouvoir être utilisée rigoureusement...

- La solution “il est interdit d’entrer en section critique” respecte cette “spécification”...
- Il faut ajouter une contrainte qui garantisse que toute demande soit satisfaite.
  - pas de *famine*
  - pas d’*interblocage* (étreinte fatale)
- Cette satisfaction peut être précisée :
  - l’ordre de satisfaction doit être le même que l’ordre de demande d’entrée
  - une certaine *équité* est requise
  - l’attente est bornée
  - ...

Famine

Définition

Aucune garantie pour un thread d'avoir l'accès à la section critique qu'il demande.

$\Rightarrow$ Algorithme d'exclusion mutuelle non équitable.

Interblocage (Étreinte fatale)

Définition

L'interblocage se produit lorsque deux processus concurrents s'attendent mutuellement. Les processus bloqués dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique.

Exemple d'interblocage

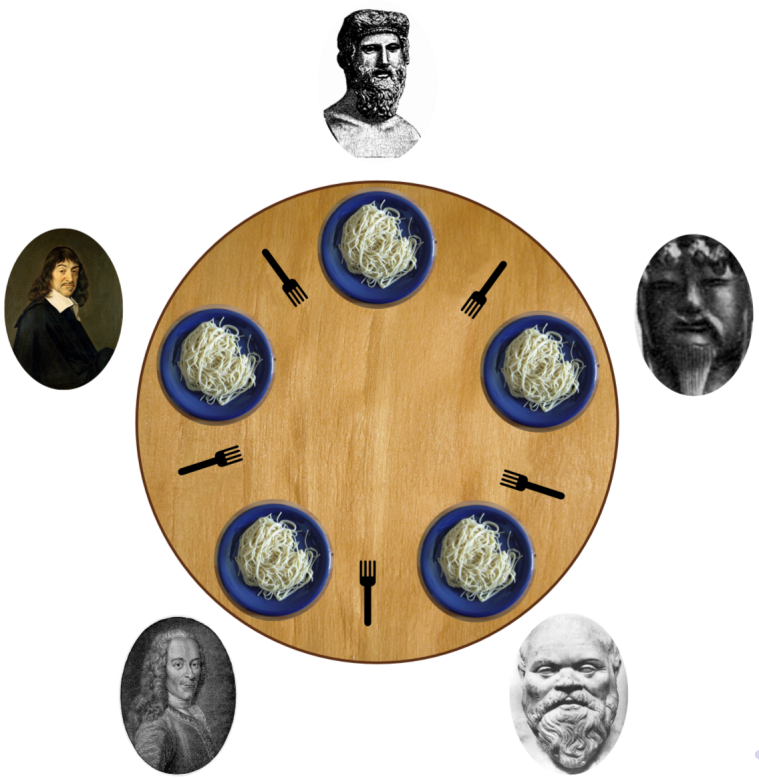
Exemple :

Thread 1	Thread 2
Obtenir M1	Obtenir M2
Obtenir M2	Obtenir M1
section critique	section critique
Rendre M2	Rendre M1
Rendre M1	Rendre M2

Le Thread 1 obtient M1.
Le Thread 2 obtient M2.
Les deux attendent la ressource obtenue par l'autre

Dîner des philosophes

- cinq philosophes, chacun a un plat de spaghetti et une fourchette à gauche
- 3 états : affamé, penseur, mangeur
- pour manger on a besoin des deux fourchettes



Spécification du Problème de l'Exclusion Mutuelle

Etant donnés des sections critiques (**SC**), un algorithme d'exclusion mutuelle doit satisfaire :

1. **exclusion mutuelle** (**EM**) : si un thread est en **SC**, aucun autre thread ne peut y être.
2. **progrès** (**P**) : si un groupe de thread demande à entrer en **SC**, l'un d'eux doit obtenir d'entrer en **SC**
3. **équité** (**E**) : tout thread demandant à entrer en **SC** doit y entrer (au bout d'un certain temps)

Sûreté et Vivacité

- **EM** est une contrainte de *sûreté*
- **P** et **E** sont des contraintes de *vivacité*
  - **P** $\Longrightarrow$ pas d'étreinte fatale
  - **E** $\Longrightarrow$ pas de famine

Les algorithmes d'exclusion mutuelle sont des *compromis* entre sûreté et vivacité...

Garantir l'Exclusion Mutuelle

Pour mettre en place de l'exclusion mutuelle, il existe de nombreux outils :

- conditions variables
- moniteurs
- sémaphores
- ...

Les Conditions Variables

Boucle sur une condition (partagée)

```
while ( condition );  // boucle infinie
                      // sur condition partagée
  /* Section Critique */
```

NB : attente active

Moniteurs

Un mécanisme supplémentaire évitant l'attente active

```
if ( condition )
    wait(); // Appel bloquant
  /* Section Critique */
notify(); //réveil d'un thread bloqué
```

Seul mécanisme avec `synchronized` pour Java $<$ 1.5

Sémaphores

Une généralisation de la notion de verrou

```
P(sem); // acquérir le sémaphore sem
  /* Section Critique */
V(sem); // libérer le sémaphore sem
```

Implémentation

Comment mettre en place ces outils et les primitives associées?
... ou “Peut-on se passer de `synchronized` en Java?”

On supposera dans la suite que la lecture et l'écriture en mémoire est atomique pour les types simples (ce qui est *presque* vrai en Java).

Première Tentative I

Une variable partagée `tour` désigne le thread dont c'est le tour :

```
static volatile Thread tour = null;
```

Cette variable est utilisée pour vérifier si l'on peut entrer en section critique ou s'il faut attendre.

```
public void Acquerir() {
    while (tour != Thread.currentThread()) {
        if (tour == null) // la voie est libre
            tour = Thread.currentThread();
    }
}
```

Première Tentative II

On remet `tour` à `null` en sortant de la section critique.

```
public void Liberer() {
  tour = null;
}
```

Chaque thread exécute une boucle infinie de tentative d'accès à la section critique :

Première Tentative III

```
public void run() {
  while(true) {
    Acquerir();
    // Section critique
    System.out.println("C'est␣le␣tour␣de␣"+
                Thread.currentThread().getName());
    try { Thread.sleep((int)(Math.random()*200));
    } catch (Exception e) {}
    System.out.println("␣␣Fin␣du␣tour␣de␣"+
                Thread.currentThread().getName());
    Liberer();
    try { Thread.sleep((int)(Math.random()*400));
    } catch (Exception e) {}
  }
}
```

Première Tentative IV

Mais ce code ne vérifie **EM**.

$\Longleftarrow$ considérer une exécution synchrone...

Deux processus I

On va commencer avec uniquement deux processus.

- `ExclusionMutuelle.java` : classe abstraite `ExclusionMutuelle`
- `Tache.java` : classe définissant des threads effectuant une boucle infinie de tentatives d'accès/sortie de section critique
- `Test.java` : définition du `main` : création des deux threads et leur démarrage.

Algorithme1.java I

```
public class Algorithme1
        extends ExclusionMutuelle{
    public Algorithme1() {
        turn = 0;
    }
    public void Pmutex(int t) {
        while (turn != t)
            Thread.yield();
    }
    public void Vmutex(int t) {
        turn = 1-t;
    }
    private volatile int turn;
}
```

Algorithme1.java II

Le problème est que la condition **P** n'est pas vérifiée : en effet, seule une exécution en tourniquet est possible.

En outre, il a été prouvé que pour n processus, il faut au moins n variables partagées.

Deux Variables : l'Etat des Processus I

```
public class Algorithme2
        extends MutualExclusion
{
    public Algorithme2() {
        flag[0] = false;
        flag[1] = false;
    }

    public void Pmutex(int t) {
        int other;
        other = 1-t;
        flag[t] = true;
        while (flag[other] == true)
            Thread.yield();
```

Deux Variables : l'Etat des Processus II

```
    }

    public void Vmutex(int t) {
        flag[t] = false;
    }

    private volatile boolean[] flag
                = new boolean[2];
}
```

Deux Variables : l'Etat des Processus III

Considérons `Algorithme2.java`

La condition **EM** n'est pas vérifiée.
Considérons l'exécution :

thread 0	thread 1	flag[t]	flag[other]
flag[1]=false?: OUI	flag[0]=false?: OUI	false	false
flag[0]:= true	flag[1]:=true	true	true
Section Critique	Section Critique	-	-

Si on initialise à `true`, alors ce sont les conditions **P** et **E** qui ne sont pas vérifiées.

```
public class Algorithme4 extends ExclusionMutuelle
{
    public Algorithme4() {
        flag[0] = false;
        flag[1] = false;
    }

    public void Pmutex(int t) {
        int other;
        other = 1-t;
        flag[t] = true;
        while (flag[other] == true) {
            flag[t] = false;
            while (flag[other])
                Thread.yield();
            flag[t] = true;
    }

    public void Vmutex(int t) {
        flag[t] = false;
    }

    private volatile boolean[] flag = new boolean[2];
}
```

Algorithme4.java

Une exécution synchrone provoque un étreinte fatale...
$\Longrightarrow$ règle de *politesse*

```
public class Dekker extends ExclusionMutuelle{
    public Dekker() {
        flag[0] = false;
        flag[1] = false;
        turn = 0;
    }
    public void Pmutex(int t) {
        int other;
        other = 1-t;
        flag[t] = true;

        while (flag[other] == true)
            if (turn == other) {
                flag[t] = false;
                while (turn == other)
                    Thread.yield();
                flag[t] = true;
            }
    }
    public void Vmutex(int t) {
        turn = 1-t;
        flag[t] = false;
    }

    private volatile int turn;
    private volatile boolean[] flag = new boolean[2];
```

Dekker (1965)

L'équité est vérifiée si l'ordonnanceur sous-jacent est équitable.

```
public class Peterson extends ExclusionMutuelle{
    public Peterson() {
        flag[0] = false;
        flag[1] = false;
        turn = 0;
    }

    public void Pmutex(int t) {
        int other;
        other = 1-t;
        flag[t] = true;
        turn = t;
        while (flag[other] && (turn == t))
            Thread.yield();
    }

    public void Vmutex(int t) {
        flag[t] = false;
    }

    private volatile int turn;
    private volatile boolean[] flag = new boolean[2];
}
```

Peterson (1981)

Cet algorithme est bien une solution au problème de l'exclusion mutuelle.

Comment s'en convaincre réellement après toutes nos précédentes et infructueuses tentatives ?

Correction d'Algorithmes Distribués

- Correction d'algorithme nécessaire
  - fiabilité (pas d'étreinte fatale)
  - sécurité ($\Longrightarrow$ pas de faille de type "race condition")
- Algorithme = objet mathématique (cf Cours LDP)
- $\Longrightarrow$ preuve formelle
- Algorithme distribué : encore plus difficile
- Méthode d'*annotation du code* Hoare (1969) puis Owicki, Gries (1975),

Principe

On associe à chaque transition du programme deux *formules logiques*. Ces formules expriment des invariants en ce point du programme.

$\{p\}$
`Instruction ;`
$\{q\}$

Si la formule p est vraie, alors après exécution de `Instruction`, la formule q doit aussi être vraie.

On infère ainsi logiquement le comportement correct du programme.

Exemple d'Annotations

G. Winskel. *The formal semantics of programming languages*. The MIT Press, 1993.

$\{x = x_0 \wedge y = y_0\}$

```
q = 0;
while (x >= y) {
```

$\{x = x_0 - qy_0 \wedge y = y_0\}$

```
  q = q+1;
```

$\{x = x_0 - (q-1)y_0 \wedge y = y_0\}$

```
  x = x-y;
```

$\{x = x_0 - qy_0 \wedge y = y_0\}$

```
}
```

$\{x = x_0 - qy_0 \wedge x < y \wedge y = y_0\}$

Non-Interférence

Attention Une fois la preuve faite pour le cas séquentiel (ie. un thread), il faut prouver que les *invariants* restent bien invariants si un *autre thread* s'exécute entre deux instructions.

- Les hypothèses sur le modèle sont prépondérantes
- Quelles sont les opérations atomiques?
- $\Longrightarrow$ il faut rajouter des variables correspondant au point de contrôle du programme.

Il s'agit en général plus de *non-interférence des preuves* que des exécutions ...

Exemple d'Annotations pour la Non-Interférence

$\{x = x_0 \wedge y = y_0 \wedge c = 0\}$

```
q = 0;
while (x >= y) {
```

$\{x = x_0 - qy_0 \wedge y = y_0 \wedge c = 1\}$

```
  q = q+1;
```

$\{x = x_0 - (q-1)y_0 \wedge y = y_0 \wedge c = 2\}$

```
  x = x-y;
```

$\{x = x_0 - qy_0 \wedge y = y_0 \wedge c = 3\}$

```
}
```

$\{x = x_0 - qy_0 \wedge x < y \wedge y = y_0 \wedge c = 4\}$

Conditions de Non-Interférences

On a

- les processeurs P_i
- la $k-$ième instruction I_k
- les préconditions pre_{ik} et postconditions $post_{ik}$ (bien entendu $post_{ik} = pre_{ik+1}$)

alors la condition de non-interférence signifie que chaque instruction I_{jl} ne peut altérer pre_{ik}.

Conditions de Non-Interférences

On doit donc vérifier, si $I_{jl} =$ `x=u;` :

- en terme de *triplet de Hoare*,

$$\{pre_{ik} \wedge pre_{jl}\}$$
`x = u ;`
$$\{pre_{ik}\}$$

- en terme logique,

$$pre_{ik} \wedge pre_{jl} \Longrightarrow pre_{ik}[x \longleftarrow u]$$

Simplification pour Peterson

Comme il y a ici une seule variable partagée on simplifie en ne numérotant pas toutes les instructions.

On introduit deux variables auxiliaires `after[0]` et `after[1]`
$\Longrightarrow$ vrai si le contrôle est après l'instruction `turn = t;` ($t = 0, 1$).
On rajoute `after[t]= true;` après celle-ci.

Notations

On note

- $I_0 = \{turn = 0\}$
- $I_1 = \{turn = 1\}$
- $preSC_0 = \{flag_0 \wedge after_0 \wedge (\neg(flag_1 \wedge after_1) \vee turn = 1)\}$
- $preSC_1 = \{flag_1 \wedge after_1 \wedge (\neg(flag_0 \wedge after_0) \vee turn = 0)\}$

Peterson pour le Thread 0

$\{\neg flag_0\}$
```
flag[0]=true; after[0]=false;
```
$\{flag_0 \wedge \neg after_0\}$
```
turn=0; after[0]=true;
```
$\{flag_0 \wedge after_0 \wedge I_0\}$
```
while (flag[1] && (turn==0)) do
```
$\{flag_0 \wedge after_0 \wedge I_0\}$
```
    Thread.yield();
```
$\{flag_0 \wedge after_0 \wedge (\neg(flag_1 \wedge after_1) \vee turn = 1)\}$
```
[ Section Critique CS0]
```
$\{flag_0\}$
```
flag[0]=false;
```

Peterson pour le Thread 1

$\{\neg flag_1\}$
```
flag[1]=true; after[1]=false;
```
$\{flag_1 \wedge \neg after_1\}$
```
turn=1; after[1]=true;
```
$\{flag_1 \wedge after_1 \wedge l_1\}$
```
while (flag[0] && (turn==1)) do
```
$\{flag_1 \wedge after_1 \wedge l_1\}$
```
    Thread.yield();
```
$\{flag_1 \wedge after_1 \wedge (\neg(flag_0 \wedge after_0) \vee turn = 0)\}$
```
[ Section Critique CS1]
```
$\{flag_1\}$
```
flag[1]=false;
```

Preuve de Correction I

- En séquentiel : OK
- Non-interférence :
  Il y a un problème à la troisième assertion
  $\Longrightarrow$ on remplace I_0 et I_1 par $I = \{turn = 0 \vee turn = 1\}$

Peterson pour le Thread 0 (version 2)

$\{\neg flag_0\}$

```
flag[0]=true; after[0]=false;
```

$\{flag_0 \wedge \neg after_0\}$

```
turn=0; after[0]=true;
```

$\{flag_0 \wedge after_0 \wedge I\}$

```
while (flag[1] && (turn==0)) do
```

$\{flag_0 \wedge after_0 \wedge I\}$

```
    Thread.yield();
```

$\{flag_0 \wedge after_0 \wedge (\neg(flag_1 \wedge after_1) \vee turn = 1)\}$

```
[ Section Critique CS0]
```

$\{flag_0\}$

```
flag[0]=false;
```

Peterson pour le Thread 1 (version 2)

$\{\neg flag_1\}$
```
flag[1]=true; after[1]=false;
```
$\{flag_1 \wedge \neg after_1\}$
```
turn=1; after[1]=true;
```
$\{flag_1 \wedge after_1 \wedge I\}$
```
while (flag[0] && (turn==1)) do
```
$\{flag_1 \wedge after_1 \wedge I\}$
```
    Thread.yield();
```
$\{flag_1 \wedge after_1 \wedge (\neg(flag_0 \wedge after_0) \vee turn = 0)\}$
```
[ Section Critique CS1]
```
$\{flag_1\}$
```
flag[1]=false;
```

Preuve de Correction I

Non-interférence :
Maintenant, seules les *preSC* contiennent des références manipulées par l'autre thread (en première, deuxième et dernière lignes)

$\rightarrow$ Il reste à vérifier la non interférence pour $preSC_0$

Preuve de Correction II

$preSC_0 = flag_0 \wedge after_0 \wedge (\neg(flag_1 \wedge after_1) \vee turn = 1)$

$\{preSC_0 \wedge \neg flag_1\}$

```
flag[1]=true; after[1]=false;
```

$\{preSC_0\}$

$\{preSC_0 \wedge (flag_1 \wedge \neg after_1)\}$

```
turn=1; after[1]=true;
```

$\{preSC_0\}$

$\{preSC_0 \wedge flag_1\}$

```
flag[1] = false;
```

$\{preSC_0\}$

Preuve de Correction III

$$preSC_0 = flag_0 \wedge after_0 \wedge (\neg(flag_1 \wedge after_1) \vee turn = 1)$$
$$preSC_1 = flag_1 \wedge after_1 \wedge (\neg(flag_0 \wedge after_0) \vee turn = 0)$$

Par symétrie, cela est aussi vrai pour le thread 1. Or

$$preSC_0 \wedge preSC_1 \Longrightarrow turn = 0 \wedge turn = 1$$

Par conséquent, on a bien l'exclusion mutuelle **EM** .

Et pour n Threads ?

Si on itère Peterson $n-1$ fois on obtient une solution à n threads. (cf `PetersonN`).

Peterson pour n Threads

```
turn of size nproc − 1
flag = [-1,-1, ... , -1] of size nproc

for(j=0; j<nproc-1; j++)
        flag[t]=j;
        turn[j]=t;
        cond = ⋁_{k=0}^{nproc−1}(flag[k] ≥ j)
        while (cond et turn[j] = t)
                yield();
```

```
public class PetersonN
{
    private volatile int[] turn;
    private int nproc;
    private volatile int[] flag;

    public PetersonN(int nb) {
        int i;
        nproc = nb;
        turn = new int[nproc-1];
        flag = new int[nproc];
        for (i=0; i < nproc; i++) {
            flag[i] = -1; }
    }
}
```

```
public class PetersonN{
    public void Pmutex(int t) {
        int j, k;
        boolean cond;
        for (j=0; j < nproc-1; j++) {
            flag[t] = j;
            turn[j] = t;
            cond = true;
            for (k=0; (k < nproc) && (k != t); k++)
                cond = cond || (flag[k] >= j);
            while (cond && (turn[j] == t))
                Thread.yield();
        }
    }

    public void Vmutex(int t) {
        flag[t] = -1;
    }
}
```