# Parallélisme

Eric Goubault
Commissariat à l'Energie Atomique
Saclay



## Plan du cours

- Présupposés; Qu'est-ce qu'un algorithme d'exclusion mutuelle?
- Premiers essais
- Algorithme de Dekker
- "Algorithme de Hyman"
- Algorithme de Perterson
- Preuve de programme; application à Peterson
- Peterson avec plus de 2 processeurs



## Algorithmes d'exclusion mutuelle:

Algorithme d'exclusion mutuelle:

- (A) si un thread se trouve en section critique, tout autre thread doit être en train d'exécuter une section non-critique.
- (B) supposons qu'un groupe de thread $I$ exécute du code dans une section non-critique, et ne requièrt pas encore d'accéder à une section critique. Supposons maintenant que tous les autres processus (groupe $J$) requièrent l'accès à une section critique, mais n'en n'ont pas encore obtenu la permission. Alors le processus qui doit rentrer en section critique doit appartenir à $J$, et cette permission ne doit pas pouvoir être reportée indéfiniment.



## Conditions et présupposés

- (C) un thread qui demande à entrer en section critique obtiendra toujours d'y rentrer au bout d'un temps fini (c'est une propriété d'*équité* qui permet d'éviter la *famine* d'un ou plusieurs processus).

Lectures et écritures en mémoire: `atomiques` (vrai pour `volatile` et types pas longs...).

## Première idée

Rajouter un champ:

```
public class CS1 extends Thread {
    Thread tour = null;
    ...  }
```

qui précise quel thread doit s'exécuter. On veut écrire une méthode:

```
public void AttendtonTour()
```

qui attend l'autorisation de s'exécuter, et

```
public void RendlaMain()
```

qui autorise les autres threads présents à essayer de s'exécuter.



## Simulation

Tous les threads instance de `CS1` ont ici la même exécution:

```
    public void run() {
        while(true) {
            AttendtonTour();
            System.out.println("C'est le tour de "+
                       Thread.currentThread().getName());
            RendlaMain();
        }
    }
```



## D'où

```
public class CS1 extends Thread {
    Thread tour = null;

    public void AttendtonTour() {
        while (tour != Thread.currentThread()) {
            if (tour == null)
                tour = Thread.currentThread();
            try {
                Thread.sleep(100);
            } catch (Exception e) {}  }  }
```



```
    public void RendlaMain() {
        if (tour == Thread.currentThread())
            tour = null;
    }

    public void run() {
        while(true) {
            AttendtonTour();
            System.out.println("C'est le tour de "+
                   Thread.currentThread().getName());
            RendlaMain();
        } }
```

```
    public static void main(String[] args) {
        Thread Un = new CS1();
        Thread Deux = new CS1();
        Un.setName("UN");
        Deux.setName("DEUX");
        Un.start();
        Deux.start();
    } }
```

Mais cela contredit (A) par exemple. Il suffit de considérer une exécution synchrone des threads et on s'aperçoit que les deux processus peuvent rentrer dans leur section critique en même temps!



## Premiers algorithmes?

```
public class Algorithme1 extends MutualExclusion
{   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;  }
```



## ExclusionMutuelle

```
public abstract class ExclusionMutuelle
{
    public static void Critique() {
        try { Thread.sleep((int) (Math.random()*3000)); }
        catch (InterruptedException e) { }  }

    public static void nonCritique() {
        try { Thread.sleep((int) (Math.random()*3000)); }
        catch (InterruptedException e) { }  }

    public abstract void Pmutex(int t);
    public abstract void Vmutex(int t);  }
```



## Tache

```
public class Tache extends Thread
{   public Tache(String n,int i,ExclusionMutuelle s) {
        name = n;
        id = i;
        section = s;  }

    public void run() {
        while (true) {
            section.Pmutex(id);
            System.out.println(name+" est en section critique");
            ExclusionMutuelle.Critique();
            section.Vmutex(id);
            System.out.println(name+" est sorti de la section critique");
            ExclusionMutuelle.nonCritique();  }  }
```

```
    private String name;
    private int id;
    private ExclusionMutuelle section;  }
```



## Test

```
public class Test
{
    public static void main(String args[]) {
        ExclusionMutuelle alg = new Algorithme1();

        Tache zero = new Tache("Tache 0",0,alg);
        Tache un = new Tache("Tache 1",1,alg);

        zero.start();
        un.start();  }  }
```



## Mais...

- `Algorithme1`, les processus ne peuvent entrer en section critique qu'à tour de rôle, ce qui est contredit (B) ( (A) et (C) sont vrais).
- En fait, on peut prouver que pour résoudre le problème de l'exclusion mutuelle entre $n$ processus dans notre cadre, il faut au moins $n$ variables partagées distinctes... (ici $n = 2$, mais seulement une variable partagée, `turn`!).
- Il faut s'arranger pour connaître l'état de chaque processus afin de prendre une décision.



## Essai...

```
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] = INIT;
        while (flag[other] == true)
            Thread.yield();  }
```

# Essai...

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

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



# Mais...

- Si on remplace `INIT` par `false` on s'aperçoit que la condition (A) d'exclusion mutuelle n'est pas satisfaite!

| zero | un | 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 d'autre part, on fait `INIT=true`, la même exécution fait que les deux processus bouclent indéfiniment, contredisant (B) et (C)!



## Autre essai...

```
public class Algorithme4 extends MutualExclusion
{   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];
    }
```

## Bon?

- Il réalise bien l'exclusion mutuelle mais peut causer un interblocage si les deux processus s'exécutent de façon purement synchrone.
- Après la première affectation, les deux tests de la boucle `while` sont vrais, d'où les deux `flag` sont mis à `false` *simultanément* puis à `true` et ainsi de suite. Les programmes de `zero` et de `un` bouclent avant la portion `Section Critique`.
- Une solution est d'instaurer une règle de *politesse* entre les processeurs.



## Dekker

```
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];  }
```



## Conclusion...

- Il réalise l'exclusion mutuelle sans blocage mutuel.
- L'équité (la condition (C)) est vérifiée si l'ordonnanceur de tâche est lui-même équitable, ce que l'on supposera.

## Hyman (1966)

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

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



```
        while (turn==other) {
            while (flag[other])
                Thread.yield();
            turn = t;  }  }

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

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

MAIS... si `turn` est à 0 et `zero` positionne `flag[1]` à `true` puis trouve `flag[0]` à `false`, alors *zero* met `flag[0]` à `true`, trouve `turn` égal à 0 et rentre en section critique. `un` affecte alors 1 à `turn` et rentre également en section critique!



## Peterson (1981)

```
public class Peterson extends MutualExclusion
{   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];  }
```

On a bien alors exclusion mutuelle sans interblocage et avec équité (pas de famine).

## Preuve

- On décore le code par des “assertions” écrites avec des prédicats logiques du premier ordre: pour un bloc $I$ on écrira $\{p\}I\{q\}$.
- Les assertions que l'on rajoute doivent être des “invariants” décrivant des relations logiques entre les valeurs des variables, vraies pour toutes les exécutions possibles (en particulier tous les ordonnancements des actions atomiques).



## Liaison avec la sémantique

- $T$ le système de transition donnant la sémantique du programme considéré. On sait qu'un état de $T$ est un (n+1)-uplet $(c_1, \ldots, c_n, E)$ où $c_i$ $(1 \leq i \leq n)$ est un point de contrôle du $i$ème processus et $E$ est un environnement.
- Alors le système d'assertion $A$ qui à chaque point de contrôle $c$ (de n'importe quel processus $i$) associe un prédicat $P$ est admissible si quelque soit l'état $(c_1, \ldots, c_n, E)$ du système de transition tel que $c_i = c$ alors $P$ est vrai pour l'environnement $E$.



## Règles de vérification

- $\{p[x \leftarrow u]\}x = u\{p\}$
- $\{p_1\}\,(if\,B\,then\,\{p_2\}I_1\{q_2\}\,else\{p_3\}I_2\{q_3\})\,\{q_1\}$ si on a $p_1 \wedge B \Rightarrow p_2$, $p_1 \wedge \neg B \Rightarrow p_3$, $\{p_2\}I_1\{q_2\}$, $\{p_3\}I_2\{q_3\}$, $q_2 \Rightarrow q_1$ et $q_3 \Rightarrow q_1$.
- $\{p_1\}while B\,\{\{p_2\}C\{q_2\}\}\,\{q_1\}$ si on a $p_1 \wedge B \Rightarrow p_2$, $p_1 \wedge \neg B \Rightarrow q_1$, $\{p_2\}C\{q_2\}$, $q_2 \wedge B \Rightarrow p_2$ et $q_2 \wedge \neg B \Rightarrow q_1$



## Exemple

```
{ x=x0 et y=y0 }
q = 0;
while (x >= y) {
  {x=x0-qy0 et y=y0}
  q = q+1;
  {x=x0-(q-1)y0 et y=y0}
  x = x-y;
  {x=x0-qy0 et y=y0}
}
{x=x0-qy0 et x<y et y=y0}
```

## Preuve de non-interférence

- Une fois la preuve faite séquentiellement, il faut prouver que les invariants trouvés pour chaque thread pris séparément, est encore valide si tout ou partie d'autres threads s'exécutent entre deux instructions, modifiant ainsi l'état local du thread!
- Pour que cela puisse se faire de façon correcte, il faut introduire des variables nouvelles dans le code séquentiel, qui déterminent le point de contrôle auquel le processus est arrivé.



## Exemple

```
{ x=x0 et y=y0 et c=0}
q = 0;
{ x=x0 et y=y0 et c=1}
while (x >= y) {
  {x=x0-qy0 et y=y0 et c=2}
  q = q+1;
  {x=x0-(q-1)y0 et y=y0 et c=3}
  x = x-y;
  {x=x0-qy0 et y=y0 et c=4}
}
{x=x0-qy0 et x<y et y=y0 et c=5}
```



## Preuve parallèle

- Chaque thread $P_i$ parmi $P_1, \ldots, P_k$ a un compteur `ci` et des assertions $A_{i,k}$ avant les instructions $I_{i,k}$ de $P_i$ (pour $k$ variant sur l'ensemble des points de contrôle de $P_i$) utilisant éventuellement des prédicats aussi sur les autres compteurs ordinaux `c1`,…,`ck`.
- Vérifier les conditions de non-interférences, quand $I_{j,l}$ est une affectation `x=u`:

$$\{(A_{i,k} \wedge A_{j,l})[x \leftarrow u]\} I_{j,l} \{A_{i,k}\}$$



## Preuve de Peterson

On n'a pas besoin ici de numéroter toutes les instructions:

- 2 variables auxiliaires `after[0]` et `after[1]` (qui servent à indiquer si le contrôle est après `turn=t` dans le code).
- + instructions `after[0]=true`, `after[1]=true` après respectivement `turn=0` et `turn=1`.
- Abbréviation `I= [turn=1 ou turn=2]`.

## Assertions

```
{ non flag[0] }
flag[0]=true; after[0]=false;
{ flag[0] et non after[0] }
turn=0; after[0]=true;
{ inv: flag[0] et after[0] et I }
while (flag[1] && (turn==0))
   { flag[0] et after[0] et I }
   Thread.yield();
{ flag[0] et after[0] et (flag[1] et (non(after[1])
                                     ou turn=1)) }
[ Section Critique CS1]
{ flag[0] }
flag[0]=false;
```



## Pour le 2ième processus...

```
{ non flag[1] }
flag[1]=true; after[1]=false;
{ flag[1] et non after[1] }
turn=1; after[1]=true;
{ inv: flag[1] et after[1] et I }
while (flag[0] && (turn==0)) do
   { flag[1] et after[1] et I }
   Thread.yield();
{ flag[1] et after[1] et (flag[0] et (non(after[0])
                                     ou turn=0)) }
[ Section Critique CS2]
{ flag[1] }
flag[1]=false;
```



## Preuve de non-interférence

```
pre(CS1) = flag[0] et after[0] et (flag[1] et
                        (non(after[1]) ou (turn=1)))
```

contient des références à des objets manipulés par l'autre processus, et ce, seulement en les lignes

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



## Mais...

```
{ pre(CS1) } flag[1]=true; after[1]=false; { pre(CS1) }
```

et

```
{ pre(CS1) } turn=1; after[1]=true; { pre(CS1) }
```

et de même par symétrie. De plus, on peut voir que

```
pre(CS1) et pre(CS2) implique (turn=0 et turn=1)
```

Ce qui implique que l'on a toujours

```
non(pre(CS1) et pre(CS2))
```

prouvant ainsi l'exclusion mutuelle.

## Peterson général

```
public class PetersonN extends ExclusionMutuelle
{
    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 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;  }
```



```
    private volatile int[] turn;
    private int nproc;
    private volatile int[] flag;  }
```



## Programme test

```
public class TestN
{   public final static int nb = 5;

    public static void main(String args[]) {
        int i;
        Tache[] liste;
        liste = new Tache[nb];
        ExclusionMutuelle alg = new PetersonN(nb);
for (i=0; i < nb; i++) {
            liste[i] = new Tache("Tache "+i,i,alg);
            liste[i].start();  }  }  }
```

## Remarques finales

- C'est la solution pour 2 processeurs itérée $n-1$ fois. `flag[i]`$= -1$ signifie que $liste[i]$ ne s'engage pas en section critique. `turn` permet de gèrer les conflits dans un couple de processus.
- Cet algorithme réalise bien l'exclusion mutuelle sans interblocage et avec équité.
- Améliorations possibles: algorithme de Burns (1981). Entièment symétrique qui minimise le nombre de variables utilisées ainsi que les valeurs qu'elles peuvent prendre en cours de programme.