Inf 431 – Cours 12

Sémaphores
Appliquettes

jeanjacqueslevy.net

secrétariat de l'enseignement:
Catherine Bensoussan
cb@lix.polytechnique.fr
Aile 00, LIX,
01 69 33 34 67

www.enseignement.polytechnique.fr/informatique/IF

Plan

Bibliographie

Mordechai Ben-Ari, *Principles of Concurrent Programming*, Prentice Hall, 1982.

J. Misra and K. M. Chandy, *Parallel Program Design : A Foundation*, Addison-Wesley, 1988.

Algorithme de Peterson (1/5)

```
class Peterson extends Thread {
  static int tour = 0;
  static boolean[ ] actif  = {false, false};
  int i, j;
  Peterson (int x) { i = x; j = 1 - x; }

  public void run() {
    while (true) {
      actif[i] = true;
      tour = j;
      while (actif[j] && tour  == j)
        ;
      // section critique
      actif[i] = false;
  } }

  public static void main (String[ ] args) {
    Thread t0 = new Peterson(0), t1 = new Peterson(1);
    t0.start(); t1.start();
} }
```

Algorithme de Peterson (2/5)

Preuve de sureté. (*safety*)

Si t_0 et t_1 sont tous les deux dans leur section critique. Alors $\mathit{actif}[0] = \mathit{actif}[1] = \mathit{true}$.

Impossible car les deux tests auraient été franchis en même temps alors que tour favorise l'un deux. Donc un seul est entré. Disons t_0.

Cela veut dire que t_1 n'a pu trouver le tour à 1 et n'est pas entré en section critique.

Preuve de vivacité. (*lifeness*)

Supposons t_0 bloqué dans le while.

Cas 1 : t_1 non intéressé à rentrer dans la section critique. Alors $\mathit{actif}[1] = \mathit{false}$. Et donc t_0 ne peut être bloqué par le while.

Cas 2 : t_1 est aussi bloqué dans le while. Impossible car selon la valeur de tour, l'un de t_0 ou t_1 ne peut rester dans le while.

Algorithme de Peterson (3/5)

- avec des assertions où on fait intervenir la ligne des programmes c_0 et c_1 (compteur ordinal) exécutée par t_0 et t_1

```
.  public void run() {
.    while (true) {
```

$$\{\neg\, actif[i] \land c_i \neq 2\}$$

```
1      actif[i] = true;
```

$$\{actif[i] \land c_i = 2\}$$

```
2      tour = j;
```

$$\{actif[i] \land c_i \neq 2\}$$

```
3      while (actif[j] && tour == j)
.        ;
```

$$\{actif[i] \land c_i \neq 2 \land (\neg\, actif[j] \lor tour = i \lor c_j = 2)\}$$

```
.      // section critique
.      ...
.      // fin de section critique
5      actif[i] = false;
```

$$\{\neg\, actif[i] \land c_i \neq 2\}$$

```
6      Thread.yield();
.  } }
```

Algorithme de Peterson (4/5)

Preuve de sureté :

- Preuve par énumération des cas (*model checking*)

- Si t_0 et t_1 sur la ligne 5, on a :

$$\begin{array}{rl} & \mathit{actif}[0] \ \wedge \ c_0 \neq 2 \ \wedge \ (\neg\, \mathit{actif}[1] \vee \mathit{tour} = 0 \vee c_1 = 2) \\ \wedge & \mathit{actif}[1] \ \wedge \ c_1 \neq 2 \ \wedge \ (\neg\, \mathit{actif}[0] \vee \mathit{tour} = 1 \vee c_0 = 2) \end{array}$$

équivaut à

$$\begin{array}{rl} & \mathit{actif}[0] \wedge c_0 \neq 2 \wedge \mathit{tour} = 1 \\ \wedge & \mathit{actif}[1] \wedge c_1 \neq 2 \wedge \mathit{tour} = 0 \end{array}$$

équivaut à

$$\mathit{tour} = 0 \wedge \mathit{tour} = 1 \wedge P$$

implique

$$\mathit{tour} = 0 \wedge \mathit{tour} \neq 0 \wedge P \ \equiv \ \text{false}$$

impossible.

Algorithme de Peterson (5/5)

Preuve de vivacité.

- Si t_0 et t_1 dans la boucle while :

$$\begin{array}{rl} & \mathit{actif}[1] \wedge \mathit{tour} = 1 \wedge \mathit{actif}[0] \wedge \mathit{tour} = 0 \\ \equiv & \mathit{tour} = 0 \wedge \mathit{tour} = 1 \wedge P \\ \equiv & \textsf{false} \end{array}$$

- Si t_0 en dehors de la boucle while et t_1 dedans, la preuve se complique car faisant intervenir l'évolution dans le temps (logique temporelle ou modale).
  Par exemple :

$$\neg\, \mathit{actif}[0] \wedge \mathit{tour} = 0 \wedge \mathit{actif}[0] \wedge c_0 = 2$$

  équivaut à

$$\neg\, \mathit{actif}[0] \wedge \mathit{tour} = 0 \wedge c_0 = 2$$

  alors le programme évolue vers $\mathit{tour} = 1$ et le tout devient faux. On quitte donc la boucle while.

Exercice 1 Généraliser l'algorithme de Peterson à n processus.

Sémaphores (1/5)

- L'algorithme de Peterson n'a qu'un intérêt théorique (idem pour l'algorithme de Dekker).
- Comme primitives de bas niveau, la littérature de la concurrence considère la notion de sémaphore `[Dijkstra 65]`.
- un sémaphore est une variable s booléenne avec deux opérations :
  - $P(s)$, prendre *Proberen* le sémaphore :
    Si s est $true$, on le met à $false$ de manière atomique ;
    sinon l'instruction attend sur s.
  - $V(s)$, libérer *Verhogen* le sémaphore :
    Si un processus attend sur s, on le réveille sans changer la valeur de s ; sinon on met s à $true$.

Sémaphores (2/5)

- A la différence des variables de condition, les sémaphores ne sont pas attachés à un verrou, mais ont une mémoire.
- En fait, ce sont les opérations effectuées à l'entrée ou à la sortie d'une section critique (synchronized en Java).

```
static Semaphore s;
while (true) {
  P(s);
  section critique
  V(s);
}
```

Exercice 2 (difficile) Programmer les variables de condition de Java avec des sémaphores.

Exercice 3 (très difficile) Programmer les variables de condition des *Posix Threads* avec les primitives de Java et des sémaphores.

Sémaphores (2/5)

- La classe `Semaphore` est fournie en Java 1.5. On peut la programmer très facilement avec des synchronized et `wait`, `notify`.

```
public class Semaphore {
  private boolean libre;
  public Semaphore(boolean x) {
    libre = x;
  }
  public synchronized void P() throws InterruptedException {
    while (!libre )
      wait();
    libre = false;
  }
  public synchronized void V() {
    libre = true;
    notify();
  }
}
```

Sémaphores (3/5)

Avec les sémaphores, on peut programmer la file d'attente concurrente de longueur 1. Au début `s_vide` = true, `s_plein` = false.

```
static void ajouter (int x, FIFO f) {
  P(s_vide);
  f.contenu = x;
  f.pleine = true;
  V(s_plein);
}

static int supprimer (FIFO f) {
  int res;
  P(s_plein);
  res = f.contenu;
  f.pleine = false;
  V(s_vide);
  return res;
}
```

Sémaphores (4/5)

Pour programmer la file de longueur n, il est plus simple de se servir de la notion de sémaphore généralisé.

- Un sémaphore généralisé a une valeur entière positive ou nulle avec deux opérations :
- (prendre la sémaphore) $P(s)$ teste $s > 0$. Si oui, on décrémente s. Sinon l'instruction attend sur s.
- (libérer le sémaphore) $V(s)$ réveille un processus en attente sur s s'il existe un tel processus. Sinon on incrémente s.

En première approximation, un sémaphore booléen est un sémaphore initialisé à 1.

Exercice 4 Montrer que la remarque précédente n'est pas tout à fait exacte.

Sémaphores (5/5)

Soit n la taille de la file. Au début, $s\_libres = n$ et $s\_occupes = 0$.

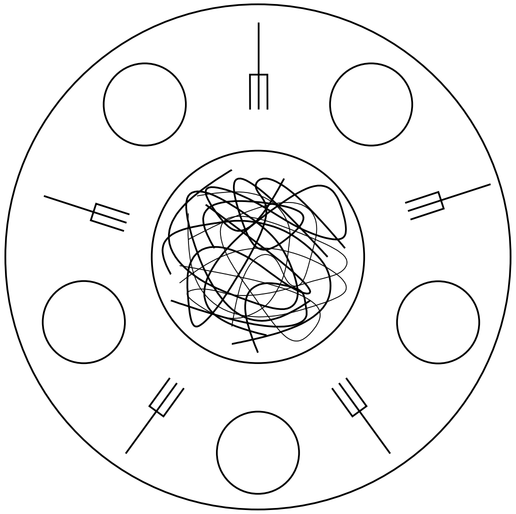
```
static void ajouter (int x, FIFO f) {
  P(s_libres);
  synchronized (f) {
    f.contenu[f.fin] = x;
    f.fin = (f.fin + 1) % f.contenu.length;
    f.vide = false; f.pleine = f.fin == f.debut;
  }
  V(s_occupes);
}

static int supprimer (FIFO f) {
  P(s_occupes);
  synchronized (f) {
    int res = f.contenu[f.debut];
    f.debut = (f.debut + 1) % f.contenu.length;
    f.vide = f.fin == f.debut; f.pleine = false;
  }
  V(s_libres);
  return res;
}
```

Problème dit du producteur - consommateur.

Les 5 philosophes (1/7)

- Problème de [Dijkstra] pour tester les primitives concurrentes : verrous, conditions, sémaphores, sémaphores généralisés, etc.
- 5 moines philosophes Φ_i pensent et mangent. Pour manger, ils vont dans la salle commune, où ils dégustent un plat de spaghettis.
- il faut deux fourchettes pour manger les spaghettis. Mais, le monastère ne dispose que de 5 fourchettes.



- Comment arriver à ce qu'aucun moine ne meure de faim ?

Les 5 philosophes (2/7)

```
class Philosophe extends Thread { // ——————————— Première solution
  static Semaphore[ ] s = new Semaphore[5];
  Philosophe (int x) { i = x; }
  int i;

  public void run() {
    while (true) {
      // penser
      Semaphore.P(s[i]);
      Semaphore.P(s[(i+1)%5]);
      // manger
      Semaphore.V(s[i]);
      Semaphore.V(s[(i+1)%5]);
  } }

  public static void main (String[ ] args) {
    for (int i = 0; i < s.length; ++i) {
      Philosophe phi = new Philosophe(i);
      phi.start();
} } }
```

Sureté, mais interblocage.

Les 5 philosophes (3/7)

```
class Philosophe1 extends Thread { // ———————— Deuxième solution
  static int[ ] f = {2, 2, 2, 2, 2};
  static ConditionPosix[ ] manger = new ConditionPosix[5];
  int i;

  Philosophe1 (int x) { i = x; }

  ...

  public static void main (String[ ] args) {
    for (int i = 0; i < f.length; ++i) {
      Philosophe1 phi = new Philosophe1(i);
      phi.start();
} } }
```

- $f[i]$ est le nombre de fourchettes disponibles pour Φ_i

Les 5 philosophes (4/7)

```
static synchronized void prendreFourchettes(int i) {
  while (f[i] != 2)
    waitPosix (manger[i]);
  -- f[(i-1) % 5]; -- f[(i+1) % 5];
}

static synchronized void relacherFourchettes(int i) {
  int g = (i-1) % 5, d = (i+1) % 5;
  ++ f[g]; ++ f[d];
  if (f[d] == 2)
    notifyPosix (manger[d]);
  if (f[g] == 2)
    notifyPosix (manger[g]);
}

public void run() {
  while (true) {
    // penser
    prendreFourchettes(i);
    // manger
    relacherFourchettes(i);
} }
```

Les 5 philosophes (5/7)

- L'invariant suivant est vérifié

$$\sum_{i=0}^{4} f[i] = 10 - 2 \times mangeurs$$

- interblocage $\Rightarrow$ $mangeurs = 0$
  $\Rightarrow$ $f[i] = 2$ pour tout i $(0 \leq i < 5)$
  $\Rightarrow$ pas d'interblocage pour le dernier à demander à manger.

- famine, si, par exemple, les philosophes 1 et 3 complotent contre le philosophe 2, qui mourra de faim.

Les 5 philosophes (6/7)

- On reprend la première solution + sémaphore généralisé $salle$
  Au début $salle = 4$
  - Pour manger, les philosophes rentrent dans la salle ;
  - il y a au plus 4 philosophes dans la salle ;
  - ils sortent de la salle après le repas ;
  - et retournent penser dans leur cellule.

```
public void run() { // ——————————— Troisième solution
  while (true) {
    // penser
    SemaphoreGen.P(salle) ; // ————— début zone critique
    Semaphore.P(s[i]);
    Semaphore.P(s[(i+1)%5]);
    // manger
    Semaphore.V(s[i]);
    Semaphore.V(s[(i+1)%5]);
    SemaphoreGen.V(salle) ; // ————— fin zone critique
} }
```

Les 5 philosophes (7/7)

- 4 philosophes au plus dans la salle $\Rightarrow$ pas d'interblocage.
- l'invariant suivant est vérifié

$$room + \text{nombre de processus dans la zone critique} = 4$$

- Si Φ_i exécute `P(s[i])`, alors il finira cette instruction.
- Si Φ_i attend indéfiniment sur `P(s[(i+1)%5])`, alors Φ_{i+1} attend indéfiniment sur `P(s[(i+2)%5])`.
- Si Φ_i exécute `P(s[(i+1)%5])`, alors il finira cette instruction.
- $\Rightarrow$ Pas de famine.

Exercice 5 Programmer cette solution des 5 philosophes en Java, avec les seuls `wait`, `notify` et `notifyAll`.
(Indication : faire une classe `Fourchette` avec les deux méthodes synchronisées `prendre` et `relacher`)

Appliquettes (1/4)

- Une appliquette (*Applet*) = sous-classe des panneaux (*Panel*) $\Rightarrow$ sous-classe des conteneurs (*Container*) de AWT.
- Une appliquette est exécutée par `appletviewer` ou appelée par un navigateur grâce à des instructions spéciales en HTML :

```
<applet code=Clock.class width=250 height=40></applet>
```

- Ses méthodes principales sont :
  - `init()` appelée au chargement de l'appliquette.
  - `start()` appelée quand l'appliquette devient visible sur l'écran.
  - `stop()` appelée quand l'appliquette devient invisible sur l'écran.
  - `destroy()` appelée quand l'appliquette devient inutilisable.

Appliquettes (2/4)

```
import java.applet.*;
import java.awt.*;
import java.util.*;

public class Clock extends Applet implements Runnable {
  private Thread t;
  private boolean threadSuspended;
  final int updateInterval = 1000;
  ...
  public void init() {
    t = new Thread (this);
    t.start();
  }

  public void start() {
    if (threadSuspended) {
      synchronized (this) {
        threadSuspended = false;
        notify();
} } } }
```

Appliquettes (3/4)

```
public void run() {
  while (true) {
    try {
      Thread.sleep(updateInterval);
      synchronized (this) {
        while (threadSuspended)
          wait();
    } }
    catch (InterruptedException e) { }
    repaint();
} }

public void stop() { threadSuspended = true; }
public void destroy() { }

public void paint (Graphics g) {
  g.setFont(new Font("Helvetica", Font.BOLD, 14));
  g.drawString (new Date().toString(), 10, 25);
}
```

Appliquettes (4/4)

- on crée un processus pour ne pas bloquer la JVM du navigateur ;
- `start` de `Applet` ≠ `start` de `Thread`
- ne pas utiliser `Thread.suspend` et `Thread.resume` (obsolètes)
  ⇒ utiliser `wait` et `notify` en faisant du *polling* dans le programme principal de l'appliquette ;
  (comme pour les interruptions)
- utiliser `repaint()` lorsqu'il y a plusieurs processus
  ⇒ `update()` ⇒ `paint()`
- tester l'appliquette avec `appletviewer` qui prend en argument un fichier HTML.
  (On voit les messages d'erreur + on isole le problème)

Exercice 6 Faire une appliquette illustrant le problème des 5 philosophes.

Synchronisation par messages (1/5)

- le modèle de la mémoire partagée est peu structuré
- il ne marche pas pour les processus coopérants à distance
- $\Rightarrow$ envoi de messages avec accusés de réception, ou totalement asynchrones
- Exemples : tous les serveurs dans un système d'exploitation pour fichiers, courrier, pages web, fenêtres, imprimantes, *shell*, calcul, noms, visages, etc.
- Modèle du client/serveur :
  - un serveur a plusieurs clients
  - les clients envoient des demandes de service au serveur
  - le serveur sert ces demandes dans l'ordre des messages.
  - pas de synchronisation car le serveur est centralisé
  - plusieurs serveurs $\Rightarrow$ programmation concurrente
  - univers symétriques sans serveurs : *Peer to Peer*

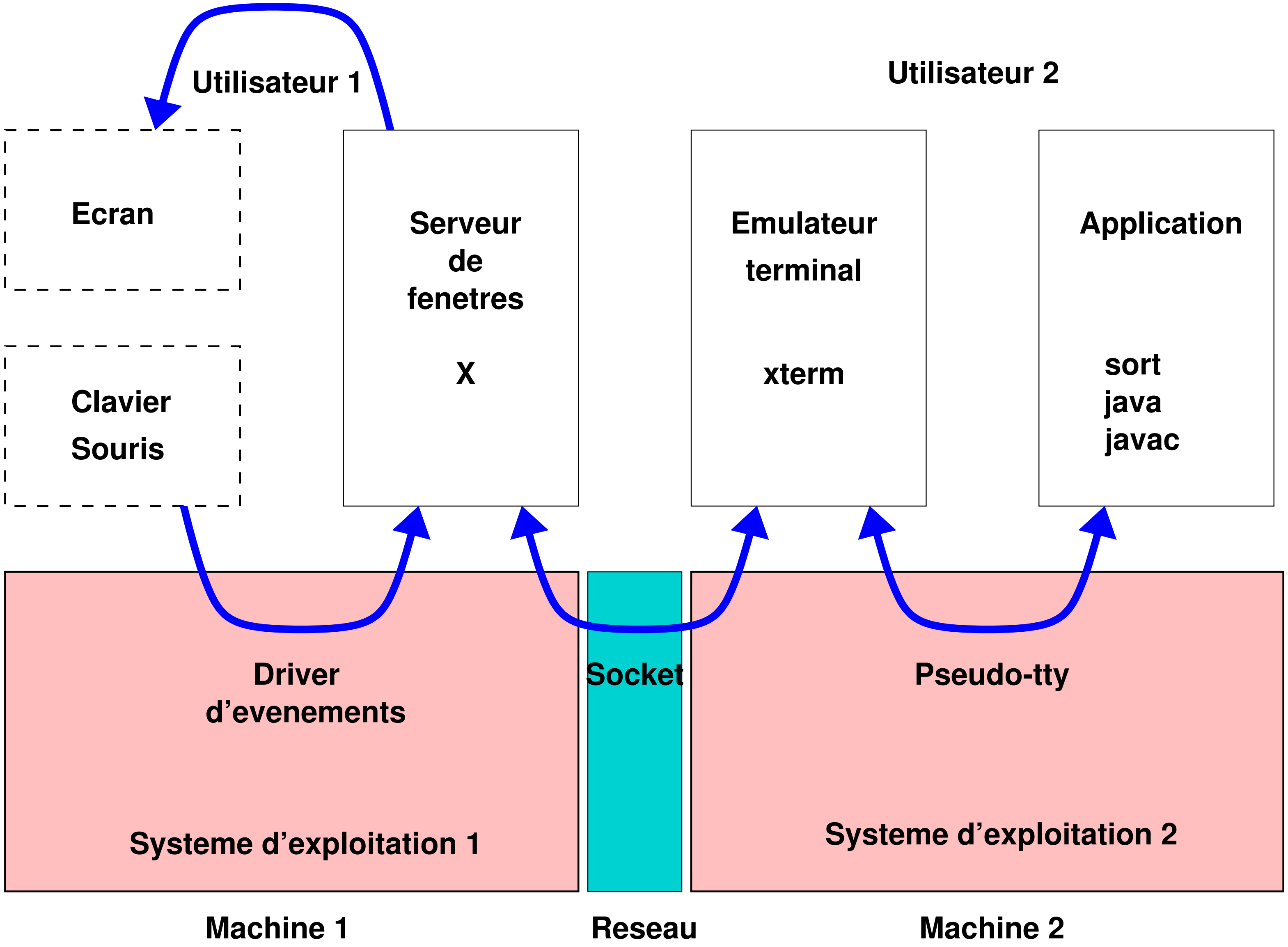
Synchronisation par messages (2/5)

- un premier exemple est NFS (*network file system*), le service de fichier à distance d'Unix.
  [Joy, 83]
- les clients montent une partition de fichiers à distance sur la hiérarchie locale.
- messages pour demandes de lectures/écritures
- la cohérence des requêtes concurrentes est assurée par le serveur qui traitent séquentiellement les diverses requêtes ⇒ modèle pauvre de la concurrence
- d'autres systèmes sont plus concurrents avec plusieurs serveurs, qui gèrent entre eux leur cohérence. Aucun système opérationnel pour le moment.
- plus généralement, la cohérence des bases de données concurrente est un problème important, et résolu (?).

Synchronisation par messages (3/5)

- un autre exemple est le serveur de fenêtres X-window [Gettys, Scheifler, 86]
- le serveur de fenêtres est en dehors du noyau système $\Rightarrow$ sureté de fonctionnement.
- le mode de connexion est cohérent avec le réseau (merci Unix BSD)
- le serveur peut ne pas être sur la même machine que les clients !
- les émulateurs terminaux sont indépendants des applications [Pike, Locanthi, 84], [Gosling, Rosenthal, 85],
- une application comme `sort`, `java`, `javac` croit parler à un terminal alpha-numérique standard.
- le serveur de fenêtres gère les événements clavier-souris et les dispatche vers l'application propriétaire du curseur.
- si le serveur de fenêtre ne sait pas dessiner un bout de fenêtre cachée, il le redemande à l'application propriétaire.

Synchronisation par messages (4/5)



cf. Cours systèmes d'exploitation et réseaux en majeure 2

Synchronisation par messages (5/5)

- théorie de la communication par rendez-vous
- très belle théorie : CSP [Hoare, 78], CCS, π-calcul [Milner, 80]. Pleins de travaux sur 20 ans.
- logiques temporelles
- $\Rightarrow$ analyseurs statiques de programme (cf. cours 14)
- la mise au point des programmes concurrents est un domaine actif de recherche.