Inf 431 – Cours 10

Processus et concurrence

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

1. Création de processus
2. Terminaison
3. Variables partagées
4. Atomicité et sections critiques
5. Interblocage
6. Famine
7. Synchronisation par variables de condition

Concurrence (1/2)

- Les programmes séquentiels exécutent séquentiellement leurs instructions.
- $\Rightarrow$ ils ne font qu'une seule chose à la fois.

Mais il faut :

- programmer des machines multi-processeurs ;
  (Cray ; Thinking Machines ; Deep Blue ; Blue Gene ; gros serveurs)
- gérer des événements asynchrones lents ;
  (entrée au terminal, signaux d'un capteur, etc)
- s'adapter au comportement concurrent des humains ;
- gérer les systèmes multi-utilisateurs ;
- réduire les délais d'attente ;
- et aller plus vite.

Concurrence (2/2)

- `x = 3; y = 25;` peut s'exécuter en parallèle
  `x = 3;` $||$ `y = 25;` sur deux processeurs
- `x = 3; y = x;` ne s'exécute pas en parallèle
  dépendances entre les deux instructions
- sur une machine vectorisée, on exécute en une seule instruction

```
for (int i = 0; i < 100; ++i)
  a[i] = b[i] + c[i];
```

- parallélisation des programmes
  automatique ou manuelle

- concurrence $\neq$ parallélisme
- concurrence = programmation concurrente + synchronisation
- parallélisme = optimisation des programmes pour aller plus vite

- parallélisme $\Rightarrow$ supercalculateurs.

Processus (1/6)

Un processus (*Thread*) est un programme qui s'exécute.

```
class Code implements Runnable {

  public void run () {
    while (true) {
      System.out.println ("Bonjour de " + Thread.currentThread());

  } }

  public static void main (String[ ] args) {
    Code p = new Code();
    Thread t1 = new Thread(p);
    Thread t2 = new Thread(p);
    t1.start(); t2.start();
} } Exécution
```

Processus (2/6)

Un processus (*Thread*) est un programme qui s'exécute.

```
class Code implements Runnable {

  public void run () {
    while (true) {
      System.out.println ("Bonjour de " + Thread.currentThread());
      Thread.yield();
  } }

  public static void main (String[ ] args) {
    Code p = new Code();
    Thread t1 = new Thread(p);
    Thread t2 = new Thread(p);
    t1.start(); t2.start();
} } Exécution
```

Processus (3/6)

En Java

- l'interface `Runnable` contient la méthode $run()$ qui sera le point d'entrée appelé par le lancement d'un processus.
  ($\simeq$ $main$ mais moins standardisé pour lui passer des arguments)
- le constructeur `Thread(p)` crée un processus à partir d'un objet `Runnable`
- le lancement d'un processus se fait en appelant sa méthode $start()$.

- dans l'exemple précédent, il y a 3 processus :
  - celui qui exécute le programme principal lancé par la JVM ;
  - t_1 et t_2 qui exécutent le même code p.
- `currentThread` est une variable statique de la classe `Thread` donnant le processus courant.

- `Thread.yield` est facultatif (mais aide l'ordonnanceur).

Processus (4/6)

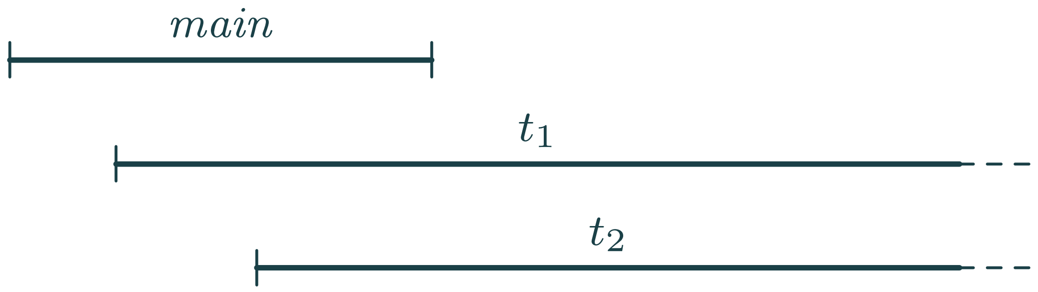
Le même exemple en programmation par objets.

```
class Code1 extends Thread {

  public void run () {
    while (true) {
      System.out.println ("Bonjour de " + this);
      Thread.yield();
    }
  }

  public static void main (String[ ] args) {
    Thread t1 = new Code1();
    Thread t2 = new Code1();
    t1.start(); t2.start();
  }
}
```
Exécution

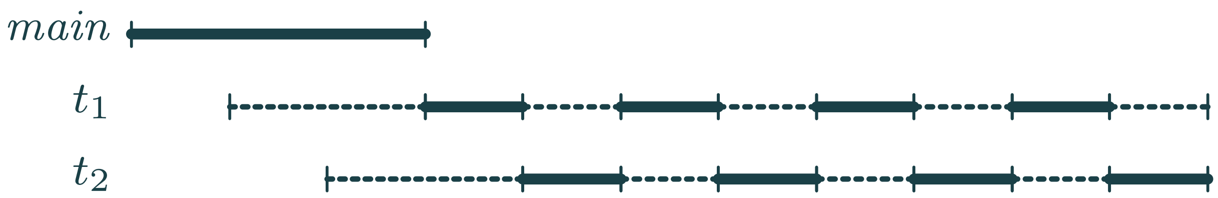
Plus simple,
mais moins de latitude dans la hiérarchie des classes.

Processus (5/6)

- Les trois processus fonctionnent en parallèle ;



- Si moins de 3 processeurs, il faut partager l'utilisation des processeurs (temps partagé) :



Processus (6/6)

- les *threads* sont des processus légers (*light-weight processes*).
- les processus (lourds) sont les processus gérés par le système d'exploitation (Unix, Linux, Windows). En Unix, on obtient leur liste par la commande **`ps`**, **`ps aux`**, **`top`**, . . .
- les processus légers sont gérés par un sous-système : ici l'interpréteur de programmes Java (la JVM).
- ils sont légers car basculer d'un processus à un autre est une opération peu couteuse (changement de contexte rapide)
- ici, comme nous ne nous soucions pas du système, nous les appelons simplement processus.
- l'exécution des processus est gérée par un ordonnanceur (*scheduler*), qui donne des tranches de temps à chaque processus, aussi équitablement que possible.
- Exemples de cercles (`Boussinot, ENSMP Sophia`) : 1 2 3 4 5

Terminaison (1/3)

```
class Exemple1 implements Runnable {

  public void run () {
    try {
      while (true) {
        System.out.println ("Bonjour de " + Thread.currentThread());
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) { }
  }

  public static void main (String[ ] args) {
    Thread t = new Thread(new Exemple1());
    t.start();
    t.interrupt();
  }
}
```

Un processus t est souvent un programme qui ne s'arrête jamais.

On peut l'interrompre : t doit alors gérer sa propre terminaison. Ainsi les invariants sont préservés.

Terminaison (2/3)

```
class Exemple2 extends Thread {

  public void run () {
    while (!isInterrupted()) {
      System.out.println ("Bonjour de " + this);
    }
  }

  public static void main (String[ ] args) {
    Thread t = new Exemple2();
    t.start();
    t.interrupt();
  }
} Exécution
```

Un processus teste périodiquement s'il n'est pas interrompu.

Terminaison (3/3)

```
class Exemple3 extends Thread {

  public void run () {
    System.out.println ("Bonjour de " + this);
  }

  public static void main (String[ ] args)
    throws InterruptedException {
    Thread t = new Exemple3();
    t.start();
    t.join(0);
  }
} Exécution
```

On peut attendre la fin d'un processus en précisant un délai maximum d'attente en argument de *join* en millisecondes ($0 =$ délai infini).

La méthode *join* lance une exception si un autre processus a interrompu le processus courant.

Variables partagées (1/2)

Une même variable x peut être modifiée par deux processus.

```
class P1 extends Thread {
  public void run () {
    while (true) {
      P.x = 23;
      Thread.yield();
} } }
```

```
class P2 extends Thread {
  public void run () {
    while (true) {
      P.x = 7;
      Thread.yield();
} } }
```

```
class P {
  static int x = 0;

  public static void main (String[ ] args)
    throws InterruptedException {
    Thread t1 = new P1(), t2 = new P2();
    t1.start(); t2.start();
    while (true) {
      System.out.println (x) ; // ———————— Valeur de x ?
      Thread.sleep (1000);
} } } Exécution
```

Variables partagées (2/2)

- on ne peut prédire l'ordre dans lequel P_1 ou P_2 écrira la variable partagée x.
- cet ordre peut changer entre deux exécutions du même programme.
- les programmes concurrents sont non déterministes.

Atomicité et sections critiques (1/5)

- l'écriture d'un scalaire sur 32 bits est atomique ;
  l'écriture d'un entier long ou d'un flottant double ne l'est pas.
- les expressions ne sont pas atomiques.
- sens de `i = 2; x[i] = i+3;` || `++i;` ? ?
- ou pour un système de réservations de places

```
class Avion extends Thread {
  static final int NBMAXSIEGES = 100;
  static boolean[ ] estOccupe = new boolean[NBMAXSIEGES];

  public void run () {
    for (int i = 0; i < NBMAXSIEGES; ++i )
      if ( !estOccupe[i] ) {
        estOccupe[i] = true;
        reserverLeSiege(i); Exécution
  } }

  static void reserverLeSiege (int i) { System.out.println (i); }
```

Atomicité et sections critiques (2/5)

- Une section critique rend atomique un ensemble d'instructions,
- on peut ainsi régler le problème de la réservation de places :

```
class Avion extends Thread {
  static final int NBMAXSIEGES = 100;
  static boolean[ ] estOccupe = new boolean[NBMAXSIEGES];

  public synchronized void run () {
    for (int i = 0; i < NBMAXSIEGES; ++i )
      if ( !estOccupe[i] ) {
        estOccupe[i] = true;
        reserverLeSiege (i);
  } }

  static void reserverLeSiege (int i) { System.out.println (i); }
```

- Le qualificatif synchronized contrôle l'accès à une section critique.

Atomicité et sections critiques (3/5)

- par exemple pour modifier plusieurs données (composites) en maintenant des invariants.

```
class ComptesBancaires {
  private int compteA, compteB;

  synchronized void transfertVersB (int s) {
    compteA = compteA - s;
    compteB = compteB + s;
  }
}
```

- La somme des deux comptes doit rester constante, même si les appels de la méthode *transfertVersB* sont faits dans deux processus distincts.
- Les sections critiques fonctionnent en exclusion mutuelle.

Atomicité et sections critiques (4/5)

- le processus qui appelle une méthode synchronized doit obtenir un verrou sur l'objet auquel elle appartient.
- si le verrou est déjà pris par un autre processus, l'appel est suspendu jusqu'à ce que le verrou soit disponible.
- en Java, les verrous sont associés aux objets. Une seule méthode synchronized peut être appelée simultanément sur un même objet. Mais une même méthode peut être appelée simultanément sur deux objets différents.
- une méthode static synchronized fonctionne avec un verrou associé à toute la classe.
- synchronized n'existe que pour les méthodes et non pour l'accès à un champ de données.
- remarque : un processus, possédant déjà un verrou, peut le reprendre sans se bloquer ($\neq$ *Threads* Posix)
  $\Rightarrow$ une méthode synchronized récursive ne se bloque pas.

Atomicité et sections critiques (5/5)

- Le qualificatif synchronized fonctionne aussi sur une instruction.

```
class ComptesBancaires {
  private int compteA, compteB;

  void transfertVersB () {
    ...
    synchronized (this) {
      compteA = compteA - s;
      compteB = compteB + s;
    }
    ...
  }
}
```

- L'argument de synchronized est l'objet sur lequel on prend le verrou. Cette forme de section critique permet de faire une section critique plus finement que sur des appels de méthodes.
- Elle permet aussi un style moins programmation par objets.

Interblocage

- Créer des sections critiques demande un peu d'attention, puisque des interblocages (*deadlocks*) sont possibles.

```
class P1 extends Thread {
  public void run () {
    synchronized (a) {
      synchronized (b) {
        ...
} } } }
```

```
class P2 extends Thread {
  public void run () {
    synchronized (b) {
      synchronized (a) {
        ...
} } } }
```

- P_1 prend le verrou sur a,
  P_2 prend le verrou sur b,
  $\Rightarrow$ interblocage
- Détecter les interblocages peut être très complexe.
  (absence d'interblocages $\simeq$ terminaison des programmes)

Exercice 1 Faire un exemple d'interblocage avec des appels sur des méthodes synchronisées.

Famine

- Rien ne garantit qu'un processus bloqué devant une section critique passera un jour dans la section critique
  $\Rightarrow$ Famine (*starvation*)
- On suppose souvent que la politique d'ordonnancement est juste et garantit de donner la main un jour à tout processus qui l'a demandée (*fair scheduling*) $\Rightarrow$ Problème de l'équité
  - Equité faible : si un processus est prêt à s'exécuter continument, il finira par s'exécuter.
  - Equité forte : si un processus est prêt infiniment à s'exécuter, il finira par s'exécuter.
- Parfois on veut le garantir logiquement en forçant un ordre de passage entre processus. (cf. cours suivant)

Conditions (1/7)

- Paradigme de la concurrence $\equiv$ la file d'attente concurrente.
- Pour simplifier supposons la file f d'au plus de longueur 1.
- *ajouter* et *supprimer* s'exécutent de manière concurrente.

```
static synchronized void ajouter (int x, FIFO f) {
  if (f.pleine)
    // Attendre que la file se vide
  f.contenu = x;
  f.pleine = true;
}

static synchronized int supprimer (FIFO f) {
  if (!f.pleine)
    // Attendre que la file devienne pleine
  f.pleine = false;
  return f.contenu;
}
```

Conditions (2/7)

- *ajouter* doit attendre que la file f se vide, si f est pleine.
- *supprimer* doit attendre que la file f se remplisse, si f est vide
- il faut relacher le verrou pour que l'autre puisse s'exécuter.

- 2 solutions :
  - ressortir de chaque méthode sans se bloquer et revenir tester la file $\Rightarrow$ attente active (*busy wait*) qui coûte cher en temps.
  - atomiquement relacher le verrou + attendre sur une condition.

- en Java, une condition est une simple référence à un objet (son adresse). Par exemple, ici la file elle-même f.
- quand on remplit la file, on peut envoyer un signal sur une condition et réveiller un processus en attente sur la condition.

Conditions (3/7)

```
static void ajouter (int x, FIFO f) throws InterruptedException {
  synchronized (f) {
    while (f.pleine)
      f.wait();
    f.contenu = x;
    f.pleine = true;
    f.notify();
} }

static int supprimer (FIFO f) throws InterruptedException {
  synchronized (f) {
    while (!f.pleine)
      f.wait();
    f.pleine = false;
    f.notify();
    return f.contenu;
} } Exécution
```

- *wait* et *notify* sont deux méthodes de la classe *Object*. Tous les objets ont donc ces deux méthodes présentes.

Conditions (4/7)

```
synchronized void ajouter (int x) throws InterruptedException {
    while (pleine)
      wait();
    contenu = x;
    pleine = true;
    notify();
}

synchronized int supprimer () throws InterruptedException {
    while (!pleine)
      wait();
    pleine = false;
    notify();
    return contenu;
} Exécution
```

- Version plus en programmation par objets.

Conditions (5/7)

```
synchronized void ajouter (int x) throws InterruptedException {
    while (pleine)
      wait();
    contenu = x;
    pleine = true;
    notifyAll();
}

synchronized int supprimer () throws InterruptedException {
    while (!pleine)
      wait();
    pleine = false;
    notifyAll();
    return contenu;
} Exécution
```

- si plus de deux processus sont en jeu, on peut utiliser la méthode *notifyAll* de la classe *Object* pour envoyer un signal à tous les processus en attente sur la condition.

Conditions (6/7)

- *notify* n'a pas de mémoire. On réveille un quelconque des processus en attente sur la condition.
- l'action relachant le verrou et de mise en attente engendrée par *wait* est atomique. Sinon, on pourrait perdre un signal émis par *notify* avant d'attendre sur la condition.
- de même *notify* est fait avant de relâcher le verrou.
- il faut utiliser un while et non un if dans la section critique. En effet, il se peut qu'un autre processus soit réveillé et qu'il invalide le test.

Conditions (7/7)

```
class FIFO {
  int debut, fin; boolean pleine, vide; int[ ] contenu;
  ...
  synchronized void ajouter (int x) throws InterruptedException {
    while (pleine)
      wait();
    contenu[fin] = x;
    fin = (fin + 1) % contenu.length;
    vide = false; pleine = fin == debut;
    notifyAll();
  }

  synchronized int supprimer () throws InterruptedException {
    while (vide)
      wait();
    int res = contenu[debut];
    debut = (debut + 1) % contenu.length;
    vide = fin == debut; pleine = false;
    notifyAll();
    return res;
  } Exécution
```

Exercices

Exercice 2 Donner un exemple précis où *notifyAll* fait une différence avec *notify*.

Exercice 3 Expliquer dans le détail pourquoi l'action *notify* doit se faire en section critique. Que se passerait-il si l'instruction était faite en dehors de la section critique ?

Exercice 4 Faire l'exemple de la file d'attente avec la représentation en liste d'éléments.

Exercice 5 Programmer des piles concurrentes.

Exercice 6 Montrer que le réveil des processus n'est pas forcément dans l'ordre premier en attente, premier réveillé.

Exercice 7 Donner un exemple où un processus peut attendre un temps infini avant d'entrer en section critique.

Exercice 8 Comment programmer un service d'attente où les processus sont réveillés dans l'ordre d'arrivée.