Programmation parallèle et concurrente : le multi-threading

Parallélisme et concurrence

- On dit que deux processus s'exécutent en parallèle lorsqu'ils s'exécutent sur des CPU différents.

- Ils sont concurrents lorsqu'ils concourent pour l'obtention d'une même ressource CPU. Leur exécution est alors entrelacée. Chacun dispose à son tour d'un quantum de temps calcul.

Thread : définition

- Un thread est un flot de contrôle à l'intérieur d'un programme. On parle de fil d'exécution, de processus léger ou même d'activité.

- Contrairement aux processus, les threads partagent le même espace d'adressage, le même environnement (variables d'environnement, fichiers,...).

- Chacun possède son propre contexte d'exécution qui comporte le pointeur sur la pile d'exécution, le compteur ordinal, ...

- Un thread possède donc moins de ressources propres qu'un processus. Sa gestion est moins coûteuse.

Exemples de threads

- La JVM (Java Virtual Machine) est un exemple de programme multi-threadé

- Un des threads de la JVM est le Garbage Collector (ramasse-miettes) qui récupère automatiquement l'espace mémoire occupé par un objet non référencé.

Exécution concurrente de threads

- La ressource CPU doit être partagée de manière équilibrée entre les différents threads.
- L'ordonnanceur gère cette répartition. Il s'exécute sous le contrôle de l'OS et de la JVM
- L'algorithme de l'ordonnanceur diffère selon les plate-formes (Unix, Windows, Macintosh)
- Pour un CPU unique, il fonctionne en accordant successivement à chaque thread un quantum de temps

Ordonnancement (1/2)

- Chaque thread (processus léger) possède une priorité propre (attribuée par défaut ou bien choisie par le programmeur). La priorité varie de 1 à 10 (par défaut 5). La méthode `setPriority(int p)` permet de fixer la priorité.

- L'ordonnanceur attribue généralement le CPU au processus de plus forte priorité. Dès qu'un quantum se termine, le processus auquel avait été attribué ce quantum, est de nouveau en compétition pour l'attribution du CPU pendant que le processus de plus forte priorité reçoit le CPU.

- A tout moment, un processus peut décider lui-même de céder le CPU pour donner une chance aux autres de s'exécuter.

Ordonnancement (2/2)

- En fait, il existe plusieurs files d'attente correspondant aux différents niveaux de priorité. Il existe aussi plusieurs politiques d'ordonnancement.

- En général, la file de plus haute priorité est d'abord explorée et tant qu'elle contient des processus, ceux ci sont privilégiés pour l'attribution du processeur. Ensuite la file de priorité inférieure est explorée et ainsi de suite.

- Pour éviter les situations de famine (un processus ne parvient jamais à obtenir le processeur), les ordonnanceurs prévoient de monter en priorité les threads de plus faible priorité de manière à leur donner une chance de passer dans l'état actif.

Exemple

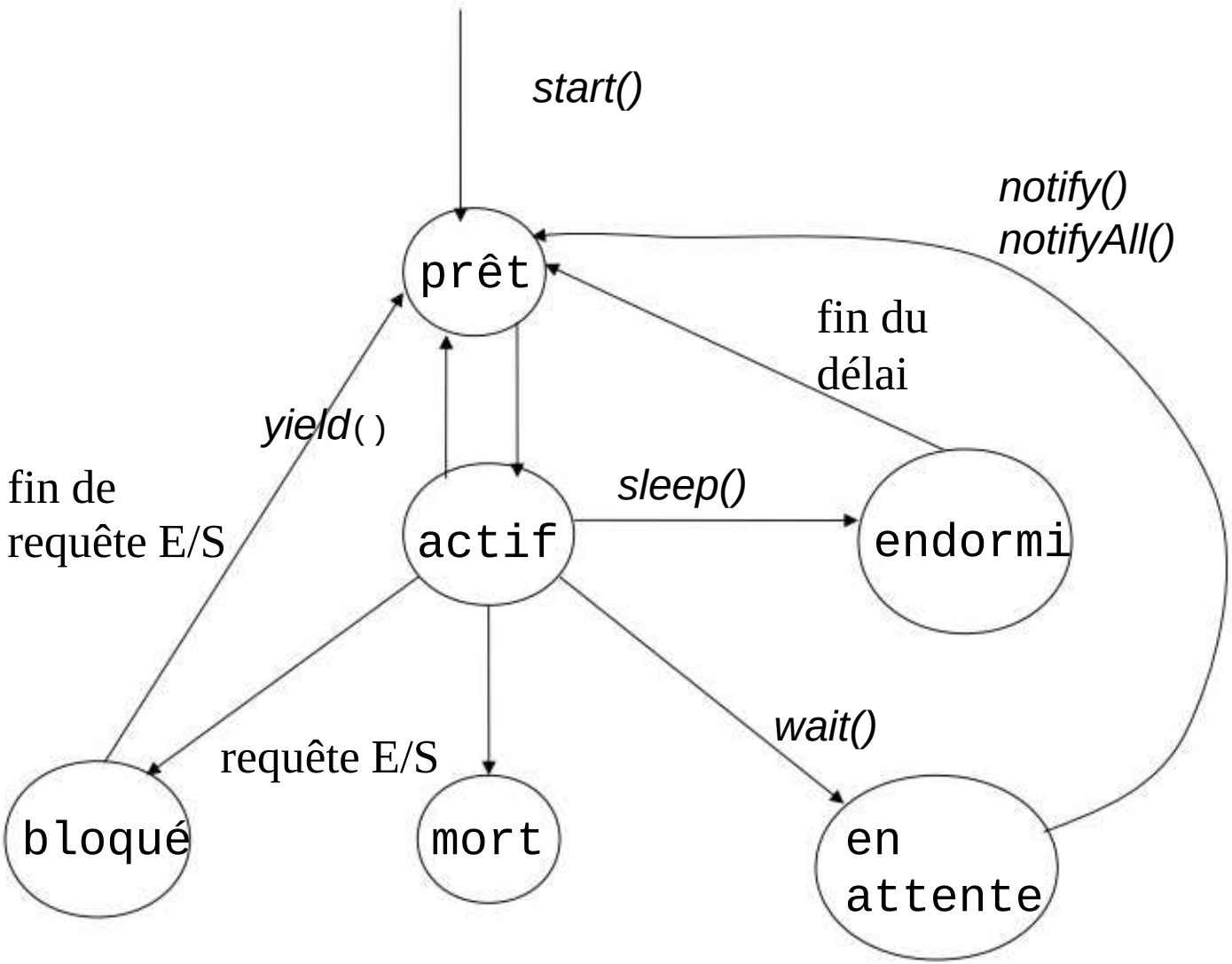
- Supposons que chaque quantum ait une durée de 1 ms
- Chaque milli-seconde, avec un processeur cadencé à 1800 megahertz, un thread peut exécuter 1800.000 instructions
- Si deux threads sont en concurrence, ils disposent chaque seconde de 500 quanta de temps, et sont donc capables d'exécuter 900 millions d'instructions

Cycle de vie d'un thread

- Ce sont des instances d'une classe dérivée de la classe `Thread`.
- Ils sont lancés par la méthode `start()`. Après sa création, un thread est dans l'état initial.

- Il reste dans cet état jusqu'à l'appel de la méthode `start()`. Il passe alors dans l'état prêt. Il est en fait placé dans une file d'attente correspondant à sa priorité.

- Lorsque le système assigne le processeur au thread (ordonnanceur), il passe dans l'état actif. Le thread exécute la méthode `run()`.


start()
notify()
notifyAll()
prêt
fin du
délai
yield()
fin de
requête E/S
sleep()
actif
endormi
wait()
requête E/S
bloqué
mort
en
attente

Contrôle d'un thread

- A l'appel de la méthode `sleep()`, le thread abandonne le CPU. Lorsque le délai d'endormissement est dépassé, le thread se range dans la file des threads prêts.
- A l'appel de la méthode `wait()`,le thread abandonne volontairement le CPU, il ne poursuivra son exécution que si un autre thread le notifie (`notify()`).
- Si un thread réalise une opération d'E/S (clavier, modem, disque, …), c'est le contrôleur du périphérique qui effectue l'opération. Pendant qu'il attend le résultat, le thread ne peut rien faire. Il est bloqué. Un autre thread peut prendre le CPU.

La classe Thread

```
public class Thread extends Object implements Runnable
{
    public Thread();
    public Thread(Runnable target);
    public Thread(String name);
    public static native void sleep(long ms)
                throws InterruptedException;
    public static native void yield();
    public final String getName();
    public final int getPriority();
    public void run();
    public final void setName();
    public final void setPriority();
    public synchronized native void start();
}
```

Exemple (1/2)

```
import java.io.*;
public class PrintNb extends Thread
{
    int nb;
    public PrintNb( int nb )
    {
        this.nb = nb;
    }
    public void run()
    {
        for ( int i=0;i<10;i++ )
            System.out.print( nb );
    }
}
```

Exemple (2/2)

```
public class Chiffres
{
    public static void main( String[] args )
    {
        Thread nb1,nb2,nb3;
        nb1= new PrintNb(1);nb1.start();
        nb2= new PrintNb(2);nb2.start();
        nb3= new PrintNb(3);nb3.start();
    }
}
```

Résultat affiché :

11111111112222222223333333333

L'interface Runnable

- Une autre manière de créer un thread est de créer une instance de `Thread` et de lui passer un objet `Runnable` qui deviendra son corps

```
public interface Runnable
{
   public void run();
}
```

- Il suffit de créer un objet qui implante la méthode `run()`

Exemple (1/3)

```
import java.io.*;
public class PrintNb implements Runnable
{
    int nb;
    public PrintNb( int nb )
    {
        this.nb = nb;
    }
    public void run()
    {
        System.out.println();
        for ( int i=0;i<10;i++ )
            System.out.print( nb );
    }
}
```

Exemple (2/3)

```
public class Nombres {
  public static void main( String[] args ){
    Thread nb1,nb2,nb3;
    nb1= new Thread( new PrintNb(1) ); nb1.start();
    nb1.setPriority( Thread.MIN_PRIORITY );
    nb2= new Thread( new PrintNb(2) ); nb2.start();
    nb2.setPriority( Thread.MAX_PRIORITY );
    nb3= new Thread( new PrintNb(3) ); nb3.start();
    nb3.setPriority( Thread.NORM_PRIORITY );
    System.out.print( "\npriorité actuelle : " );
    System.out.println( Thread.currentThread().getPriority());
    System.out.print("nombre de threads:"+Thread.activeCount());
  }
}
```

priorité actuelle : 5
nombre de threads : 7
2222222222
3333333333
1111111111

Exemple(3/3)

- On souhaite endormir un thread pendant un délai aléatoire. On réécrit la méthode `run()`.

```
public void run(){
   for(int i=0;i<10;i++){
   try{
       Thread.sleep((long)( Math.random()*1000) );
    }
    catch(InterruptedException e){
          System.out.println( e.getMessage() );}
      System.out.print( nb );
} }
```

Un résultat possible :

```
31132323122311321322312321211
```

Pourquoi synchroniser des threads ?

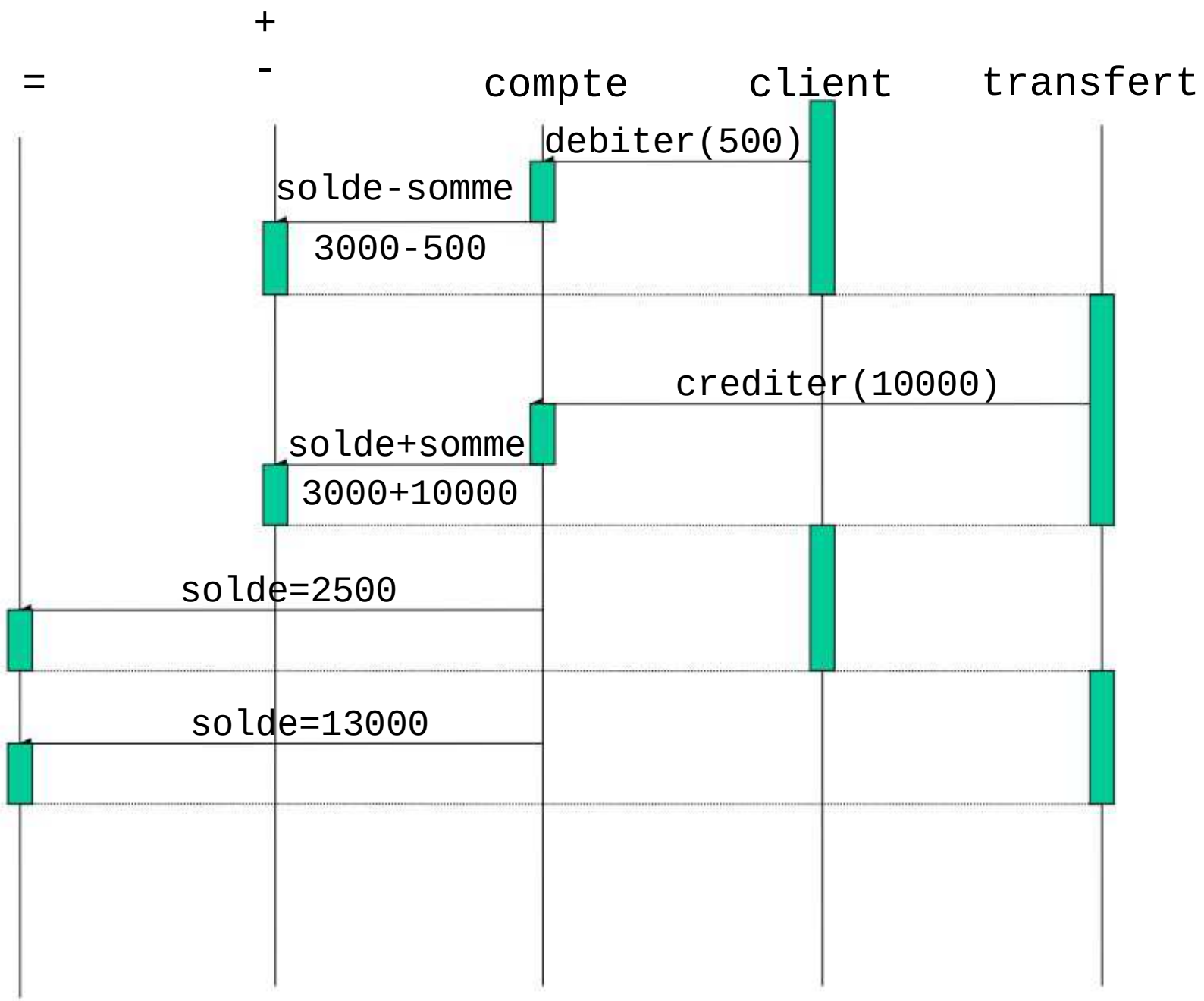
- Plusieurs threads utilisent un même objet de manière concurrente. Les méthodes de la classe `Compte` peuvent être appelées par des threads.

```
public class Compte{
   private double somme;
   public void crediter( double somme ){
      solde = solde+somme;}
   public void debiter( double somme ){
      if( solde < somme )
     throw new SoldeInsuffisant();
     else
    solde = solde-somme;
} }
```

Etude d'un cas (1/3)

```
Compte compte = new compte();
```

- Le thread `client` retire 500€ d'un guichet sur le compte `compte`
- Le thread `transfert` réalise un transfert de 10000€ sur l'objet `compte`
- Initialement le solde de l'objet `compte` est de 3000€


+
-
=
compte
client
transfert
debiter(500)
solde-somme
3000-500
crediter(10000)
solde+somme
3000+10000
solde=2500
solde=13000

Etude d’un cas (3/3)

Le langage garantit l’atomicité en lecture et écriture des variables des types primitifs comme `byte, char, short, int, float`, référence (`Object`) mais ce n’est pas le cas pour les types `long` et `double`

Le "bytecode" correspondant à l'opération `solde = solde - somme` est :

```
load_1 solde
load_2 somme
sub 1,2
store solde
```

C'est au niveau des instructions du byte code que l'exécution a lieu en exclusion mutuelle. Un entrelacement des instructions du byte code est alors possible

`=> nécessité de créer des sections critiques de code java`

Dans l'exemple, les méthodes `crediter` et `debiter` doivent être exécuter sans risque d'être interrompues

Synchronisation des threads : exclusion mutuelle

- Lorsque des threads partagent certaines données de l'application, il y a nécessité de conserver une certaine cohérence sur les données partagées => synchronisation des threads
- Tout objet est muni d'un verrou qui peut être ouvert ou fermé
- Un thread peut fermer le verrou d'un objet. Il est alors le seul à pouvoir rouvrir le verrou.
- Exemple : si le thread `t1` doit exécuter la suite d'instructions `b1` en section critique, un thread `t2` doit se mettre en attente pendant ce temps pour ne pas interférer pendant l'exécution de `b1`. Ils doivent donc se synchroniser sur un objet `obj`.
- `t1` tente d'exécuter `b1` : 2 cas :
  - le verrou de `obj` est ouvert => `t1` verrouille `obj`, exécute `b1` et libère le verrou
  - le verrou de `obj` est fermé => `t1` est mis en attente jusqu'à l'ouverture du verrou

Synchronisation des threads

- Si la synchronisation a lieu au niveau d'une méthode (et non d'une suite d'instructions), c'est l'objet représenté par `this` qui détient le verrou.

```
synchronized m(){...}
```

- Si on invoque `obj.m();`, la méthode est synchronisée sur `obj`
- Pour synchroniser un bloc d'instructions sur un objet `obj`, on déclare une section critique :

```
synchronized(obj){...}
```

Classe avec méthodes synchronisées

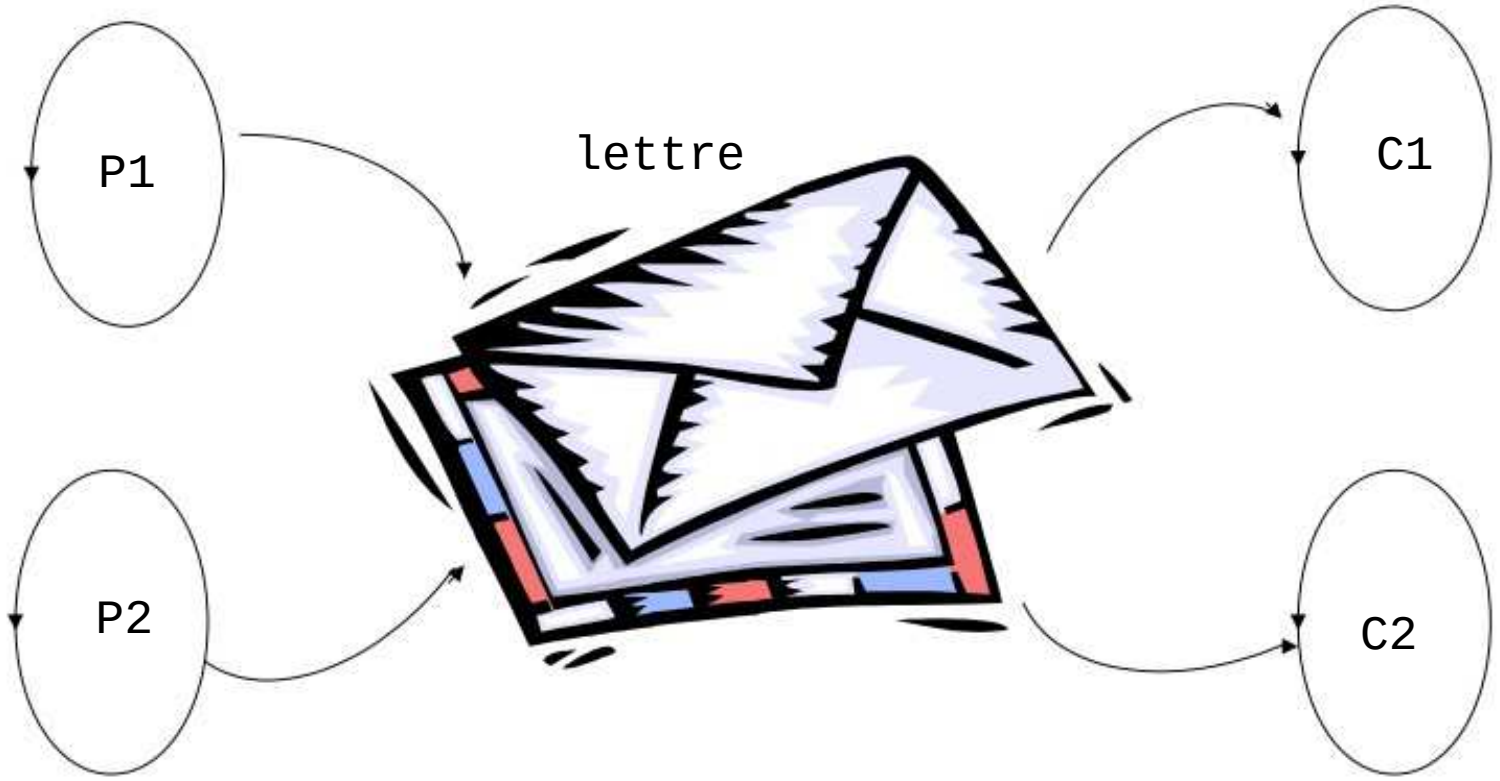
```
public class Compte{
   private double somme;
   public synchronized void crediter( double somme ){
      solde = solde+somme;}
   public synchronized void debiter( double somme ){
      if( solde < somme)
       throw new SoldeInsuffisant();
      else
       solde = solde-somme;
  }
}
```

Communication entre threads

- Un thread peut produire des résultats qui serviront de données à un autre thread
- On utilise alors un tampon intermédiaire
- Le thread producteur dépose ses résultats dans le tampon
- Le thread consommateur collecte ses données dans ce même tampon

Communication sans synchronisation

- Deux classes de threads (`Producteur` et `Consommateur`) tentent d'accéder une ressource commune (`BoiteAuxLettres`).
- Les producteurs ont pour mission de déposer des valeurs dans une variable partagée (`lettre`) et les consommateurs de les retirer.


P1
P2
lettre
C1
C2
Producteurs
Boîte aux lettres
Consommateurs

```
public class BoiteAuxLettres {
    private String lettre;

    public String retirer( String destinataire ){
     System.out.println( destinataire+" lit "+lettre );
     return lettre;
    }

    public void deposer( String lettre ){
    this.lettre = lettre;
     System.out.println( "dépot de : "+lettre );
    }
}
```

```
public class Producteur extends Thread
{
    BoiteAuxLettres boite;
    String nom;
    public Producteur( BoiteAuxLettres boite,String nom )
    {this.boite = boite;this.nom = nom;}
    public void run()
    {
        for ( int cpt = 1; cpt <5 ; cpt++)
        {
            try
            {
                Thread.sleep((int)(Math.random()*2000));
            }
            catch(InterruptedException e)
            {
                System.err.println(e.toString());
            }
            boite.deposer( nom+" ,lettre "+ cpt  );
       }
    }
}
```

```
public class Consommateur extends Thread
{
    BoiteAuxLettres boite;
    String nom;
    public Consommateur( BoiteAuxLettres boite,String nom )
    {this.boite = boite;this.nom = nom;}
    public void run()
    {
        for ( int cpt = 1; cpt <5 ; cpt++)
        {
            try
            {
                Thread.sleep((int)(Math.random()*2000));
            }
            catch(InterruptedException e)
            {
                System.err.println(e.toString());
            }
            boite.retirer( nom );
        }
   }
}
```

```
public class Client{
    public static void main( String[] args )
    {
        BoiteAuxLettres b = new BoiteAuxLettres();
        Producteur p1 = new Producteur( b, "Alex" );
        Producteur p2 = new Producteur( b, "Leo" );
        Consommateur c1 = new Consommateur( b, "Marie" );
        Consommateur c2 = new Consommateur( b, "Lucie" );
        p1.start(); p2.start();
        c1.start(); c2.start();
   }
}
```

Résultat

Comme les threads ne sont pas synchronisés, il se peut que :

- certaines données soient perdues (non consommées) si le producteur place une nouvelle donnée avant que le consommateur ne la consomme.
- de même, plusieurs consommations de la même donnée peuvent survenir avant une nouvelle production.

Lucie lit null
Lucie lit null
Marie lit null
dépot de : Alex ,lettre 1
dépot de : Alex ,lettre 2
Lucie lit Alex ,lettre 2
dépot de : Alex ,lettre 3
dépot de : Leo ,lettre 1
Marie lit Leo ,lettre 1
dépot de : Leo ,lettre 2
Marie lit Leo ,lettre 2
dépot de : Leo ,lettre 3

Communication avec synchronisation (1)

```
public class BoiteAuxLettres
{
    private boolean ok = false;
    private String lettre;

    public synchronized String retirer(String destinataire)
    {
     try{
       while( !ok ) wait();
     }catch( InterruptedException e )
     {
        System.out.println( "interruption" );
        System.exit(1);
      }
     System.out.println(destinataire+" lit "+lettre);
     ok = false;
     notifyAll();
     return lettre;
    }
```

```
public synchronized void deposer( String lettre )
{
  try
  { while( ok ) wait(); }
  catch( InterruptedException e )
  {
    System.out.println(" interruption" );
    System.exit(1);
  }
  this.lettre = lettre;
  System.out.println( "dépot de : "+lettre );
  ok = true;
  notifyAll();
}
}
```

dépot de : Alex ,lettre 1
Marie lit Alex ,lettre 1
dépot de : Leo ,lettre 1
Lucie lit Leo ,lettre 1
dépot de : Leo ,lettre 2
Marie lit Leo ,lettre 2
dépot de : Alex ,lettre 2
Lucie lit Alex ,lettre 2
dépot de : Leo ,lettre 3
Marie lit Leo ,lettre 3
dépot de : Alex ,lettre 3
Lucie lit Alex ,lettre 3

`wait` et `notify`

- Lorsque `wait()` est appelé, l'exécution du thread est momentanément interrompue tandis que le verrou est levé. D'autres threads peuvent exécuter une méthode synchronisée.
- Lorsque `notify()` est appelé, un signal est généré vers un thread en attente indiquant sa libération. Celui-ci peut donc poser le verrou et devenir à nouveau éligible
- Le thread en attente reprendra son exécution après un `notify();`
- `wait()` est toujours dans un bloc `try` car il peut lever une exception
- Si plusieurs threads sont bloqués sur un `wait(),` ils peuvent être libérés globalement par un `notifyAll()`

Communication avec gestion circulaire de tampon

- La vitesse d'exécution des threads n'étant pas la même, pourquoi faire attendre :
  - un producteur s'il est en mesure de déposer une valeur
  - un consommateur si d'autres valeurs à consommer sont présentes.

- Un tampon géré circulairement comme une file permettra à un producteur de déposer en queue une nouvelle valeur si cette file n'est pas pleine. Un consommateur pourra retirer une valeur de cette file à condition qu'elle ne soit pas vide. Ainsi, plusieurs productions (ou consommations) successives pourront avoir lieu et toutes les valeurs produites seront consommées une et une seule fois.

- Seule la classe `BoiteAuxLettres` est modifiée.

Exemple (1/2)

```
public class BoiteAuxLettres {
    private String[] lettres;
    private int tete = 0;private int queue = 0;
    private boolean ecriturePossible = true;
    private boolean lecturePossible = false;
    private static final int TAILLE = 5;

    public synchronized String retirer( String destinataire ){
    try{ while( !lecturePossible ) wait(); }
      catch( InterruptedException e ){
    System.out.println( "interruption" );System.exit(1);
    }
      String lettre = lettres[tete];
      tete = (tete+1)%TAILLE;
      if( queue==tete ) lecturePossible = false;
      System.out.println( destinataire+" lit "+lettre );
      notifyAll();
      return lettre;
    }
```

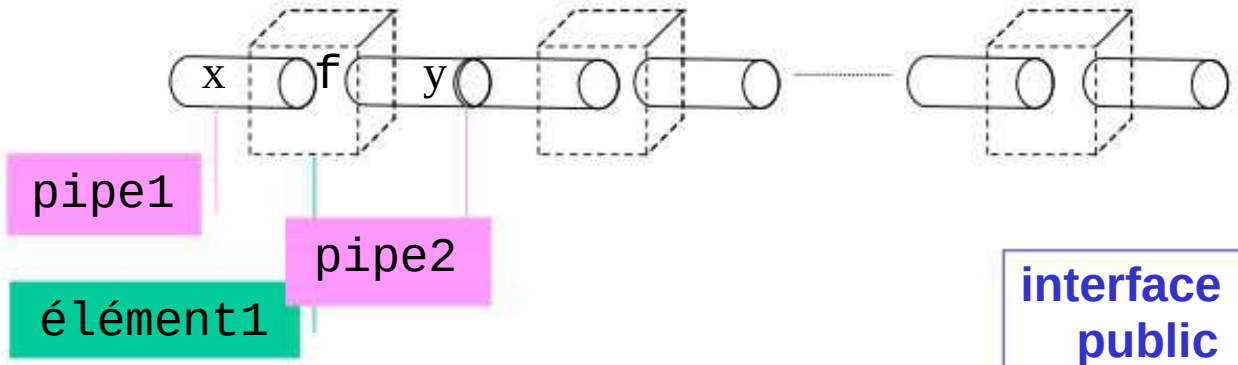
Exemple (2/2)

```
public synchronized void deposer( String lettre )
{
  try
  { while( !ecriturePossible ) wait(); }
  catch( InterruptedException e )
  {
    System.out.println(" interruption" );
    System.exit(1);
  }
  lettres[queue] = lettre;
  lecturePossible = true;
  queue = (queue+1)%TAILLE;
  if ( queue==tete ) ecriturePossible = false;
  System.out.println( "dépot de : "+lettre );
  notifyAll();
}
}
```

```
dépot de : Leo ,lettre 1
dépot de : Alex ,lettre 1
Marie lit Leo ,lettre 1
Lucie lit Alex ,lettre 1
dépot de : Leo ,lettre 2
dépot de : Alex ,lettre 2
Lucie lit Leo ,lettre 2
dépot de : Leo ,lettre 3
Marie lit Alex ,lettre 2
Marie lit Leo ,lettre 3
dépot de : Alex ,lettre 3
Lucie lit Alex ,lettre 3
```

```
dépot de : Leo ,lettre 1
Lucie lit Leo ,lettre 1
dépot de : Leo ,lettre 2
Marie lit Leo ,lettre 2
dépot de : Alex ,lettre 1
Marie lit Alex ,lettre 1
dépot de : Leo ,lettre 3
Lucie lit Leo ,lettre 3
dépot de : Alex ,lettre 2
Marie lit Alex ,lettre 2
dépot de : Alex ,lettre 3
Lucie lit Alex ,lettre 3
```

Réseau de threads en pipeline

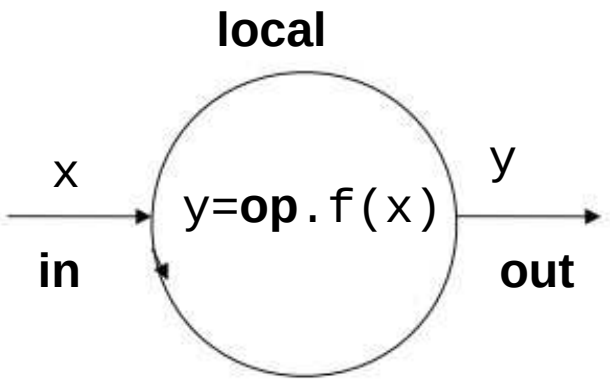



```
interface Operation{
  public int f(int x);
}
```

Chaque élément est un Thread instance de la classe `Element`
Il réalise la fonction `f`, soit `y=f(x)`,spécifiée par l'interface `Operation`
Les pipes, instances de la classe `Canal` fonctionnent en Rendez-vous.
Pour pouvoir lire (écrire), un canal de sortie (d'entrée) attend qu'un canal d'entrée (sortie) ait déposé une donnée

La classe Element

```
class Element implements Runnable{
private Canal in,out;
 private Thread local;
 private Operation op;
 Element(Canal in,Canal out,Operation op){
 this.in = in;
 this.out = out;
 this.op = op;
 local = new Thread(this); local.start();
 }
 public void run(){
  while(true){
   int x = in.lire();
   int y = op.f(x);
   out.ecrire(y);
  }
 }
}
```



La classe Canal

```
public class Canal {
private int val = 0;
 private boolean emetteur=false,recepteur=false;
 public synchronized int lire(){
  recepteur = true;
  if (!emetteur){ // en train d'écrire ou donnée indisponible
  try{ wait(); }catch(Exception e){}
  }
  recepteur = false;
  notify();
  return val;}
 public synchronized void ecrire(int x){
 val = x;
  emetteur = true;
  notify();
  if (!recepteur){ //en train de lire
  try{ wait(); }catch(Exception e){}
  }
  emetteur = false;
}}
```

La classe `PipeLine`

```
class Pipeline{
 private Canal pipe[];
 private int size;
 public Pipeline(int size, Operation op){
 pipe = new Canal[size];
 for(int i=0;i<size;i++){
 pipe[i] = new Canal();
 }
 for(int i=0;i<size-1;i++){
  new Element(pipe[i],pipe[i+1],op);
 }
 this.size = size;
}
 public void envoyer(int val){
 pipe[0].ecrire(val);
 }
 public int recevoir(){
  return pipe[size-1].lire();
 }}
```

La classe TestPipeLine

```
public class TestPipeline{
 public static void main(String args[]){
 Pipeline pipe = new Pipeline(30,new Inc());
 pipe.envoyer(5);
 int val = pipe.recevoir();
 System.out.println("pipe1, valeur recue : " + val);
 }
}
```

```
class Inc implements Operation{
public int f (int x){
   return x+1;
 }
}
```

Communication par objet partagé

Exemple : course de threads

- chaque thread incrémente un objet partagé
- le thread qui finit la course affichera toujours une valeur < 2000000
- le résultat n'est jamais le même :

```
thread 1 lancé
thread 2 lancé
thread 1 termine avec 1250408
thread 2 termine avec 1423299
```

course de threads

```
public class Course
{
   public static void main( String[] args )
   {
       Compteur c=new Compteur( 0 );
       Coureur s1=new Coureur( 1,c );
       Coureur s2=new Coureur( 2,c );
       s1.start();
       s2.start();
   }
}
```

```
public class Compteur
{
    private int valeur;
    public Compteur( int valeur ){ this.valeur = valeur; }
    public void inc(){ valeur=(valeur+1)%1000000; }
    public int get(){ return valeur; }
}
```

```
public class Coureur extends Thread
{
    private int id;
    private Compteur compteur;
    public Coureur(){}
    public Coureur( int id, Compteur compteur )
    { this.id = id;this.compteur = compteur; }

    public void run()
    {
       System.out.println( "thread "+id+" lancé" );
       System.out.flush();
       for( int i=0;i<1000000;i++ )
            compteur.inc();
       System.out.println
         ("thread "+id+" termine avec "+ compteur.get());
   }
}
```

Course de threads : commentaires

```
thread2 lancé
thread1 lancé
thread1 termine avec 784265
thread2 termine avec 915751
```

Quelques clicks sont perdus. Le "bytecode" correspondant à l'instruction
`valeur = (valeur+1)%1000000` est :

```
load valeur
load 1
add
load 1000000
mod
store valeur
```

Or chaque thread a son propre contexte d'exécution

course de threads (fin)