Université Gaston Berger de Saint-Louis

Programmation concurrente et parallèle

Pr. Ousmane THIARE

M2RSD
[www.ousmanethiare.com]

12 septembre 2025

Contenu

Contenu

En programmation séquentielle, un programme est décomposé en sous-programmes (procédures ou fonctions). Chaque sous-programme correspond à une suite d'instructions, et l'exécution du programme voit ces instructions être exécutées les unes à la suite des autres. Les premiers ordinateurs (et leurs successeurs durant quelques années) ont fonctionné selon ce mode d'exécution sérielle. Une tâche pouvait en appeler une autre, mais la tâche appelante ne continuait son exécution qu'après la terminaison de la tâche appelée. Cette approche, simple à mettre en oeuvre en comparaison des systèmes actuels, souffre toutefois de quelques faiblesses. Un seul programme ne peut être exécuté à la fois, ce qui interdit entre autres à un utilisateur de travailler sur un document en même temps qu'il compile un code source. Si le programme en cours d'exécution attend que l'utilisateur effectue une action (clic souris par exemple), alors le processeur se retrouve sous-exploité. Et enfin, si un programme engendre une erreur, et se fige, alors le système risque de se retrouver totalement bloqué.

A l'heure actuelle, les systèmes informatiques sont multi-processus, et supportent l'exécution pseudo-parallèle de plusieurs processus (programmes en exécution). Ceci a été rendu possible par la mise au point de systèmes d'exploitation nettement plus complexes. Un processeur peut donc voir plusieurs processus en cours d'exécution se partager le temps de traitement. Et de même, un processus peut être décomposé en sous-processus ou *tâche* (*threads* en anglais). Ces threads permettent de ne plus être liés à une exécution totalement sérielle des instructions. Il est dès lors possible d'avoir, par exemple, un thread responsable d'exécuter un calcul lourd pendant qu'un autre gère les interactions avec l'utilisateur. Un serveur FTP, par exemple, peut plus facilement gérer plusieurs connexions simultanées, chaque connexion étant gérée par un thread.

Contenu

Le terme *programmation concurrente* ne doit pas être confondu avec celui de *programmation parallèle* (ou programmation répartie). La programmation concurrente, telle que nous l'entendrons, se réfère à un système décomposé en tâches pouvant être exécutées dans un ordre quelconque. Certaines tâches sont exécutées avant d'autres, et certaines le sont en parallèle. La programmation parallèle traite, quant à elle, de l'exécution simultanée de tâches sur différents processeurs. Il s'agit alors de pouvoir synchroniser des processus entre eux, ceci principalement au travers d'une mémoire partagée ou de liaisons réseau.

Dans le cadre de ce cours, nous ne traiterons pas de la communication inter-processus, mais bien de la communication intra-processus. Un programme sera décomposé en threads, qui sont donc ses fils d'exécution pseudo-parallèles. Les problèmes qui vont nous intéresser sont donc le partage de données et la synchronisation entre threads. Il est intéressant de noter que le concept de programmation concurrente est autant valable sur un processeur simple coeur que sur un multi-coeur. Sur un simple coeur, les parties de threads s'exécutent tour à tour de manière transparente, et sur un multi-coeur, un réel parallélisme peut être observé, chaque coeur pouvant exécuter un ensemble de threads.

L'illustration de la programmation concurrente sera faite par le biais de 2 languages. Le premier, le langage C, n'offre dans sa définition aucun mécanisme concurrent. Pour contourner ce problème, nous allons utiliser la bibliothèque standard *pthread*, qui propose un ensemble de mécanismes pour la gestion des threads. Elle permet de décrire une application sous forme d'un ensemble de threads, ces threads étant ensuite exécutés sur la machine cible. Le second langage, Java, quant à lui, met à disposition la notion de threads et quelques mécanismes rudimentaires.

Contenu

Un processus correspond à un fichier exécutable en cours d'exécution sur un processeur. Il est entre autre caractérisé par un code (le programme à exécuter), une pile et un tas qui permettent de stocker les données nécessaires à son bon fonctionnement, un identifiant unique, et une priorité. La priorité permet au système d'exploitation de gérer plusieurs processus en cours d'exécution sur un processeur, un processus à plus haute priorité se voyant attribuer un temps d'utilisation processeur plus important.
Un processus est créé lorsqu'un autre processus lance son exécution. Nous pouvons distinguer le processus parent (celui qui lance), et le processus enfant.

La figure 1 illustre les différentes étapes de la vie d'un processus, pour un système d'exploitation de type Unix. L'état initial d'un processus est *terminé* (*destroyed* en anglais). Après sa création, il passe à l'état *prêt* (*ready* en anglais) lorsque toutes les ressources à son bon fonctionnement ont été réquisitionnées. Le système d'exploitation peut ensuite le faire passer dans l'état *élu*, (*active* en anglais), état dans lequel le processus s'exécute. Ce passage n'est pas du ressort du processus, mais bien de l'ordonnanceur, qui s'occupe d'allouer le processeur aux différents processus concurrents. A tout instant l'ordonnanceur peut replacer le processus dans l'état *prêt*, pour laisser un autre processus s'exécuter.

Il s'agit de la *préemption* d'un processus, qui se fait sans que le processus préempté n'en soit conscient. Depuis l'état *élu*, le processus peut aussi se retrouver dans l'état *bloqué* (*blocked* en anglais), lors de l'attente d'un événement ou du relâchement d'un mutex, par exemple. Il n'y a ensuite qu'une possibilité pour sortir de l'état *bloqué*. Il s'agit de réveiller le processus suite au relâchement d'un mutex ou au fait qu'un événement sur lequel le processus attend a été déclenché. Dans ce cas, le processus passe à l'état *prêt*, prêt à continuer son exécution. Lorsque le processus s'exécute, il peut se terminer. Les deux flèches sortantes de l'état *élu* représentent deux scénarios. Premièrement, si le processus a été détaché, c'est-à-dire qu'il est sans lien parental, lors de sa terminaison, le processus passe directement dans l'état *terminé*. Deuxièmement, si la terminaison du processus est importante pour le reste de l'application, alors il passe dans l'état *zombie* (*zombied* en anglais). Il y reste jusqu'à ce que le processus parent effectue une *jointure*, c'est-à-dire qu'il récupère les informations retournées par le processus en cours de terminaison.

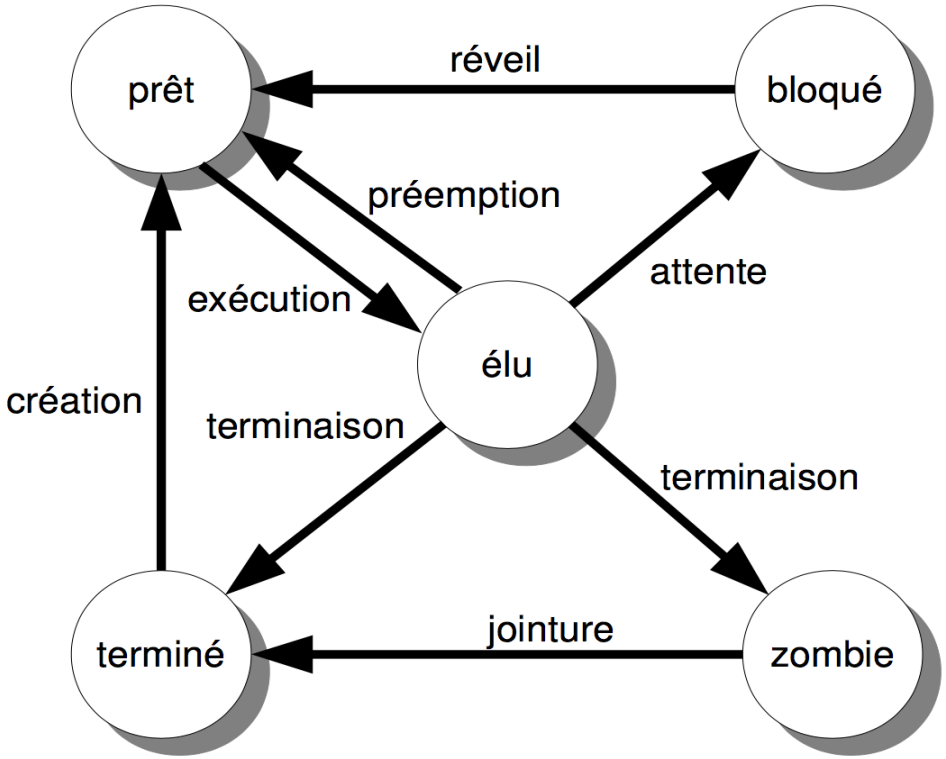


Figure: Etats et transitions d'un processus Unix

La figure 2 suivante illustre la décomposition de l'espace d'adressage d'un processus en trois parties principales :

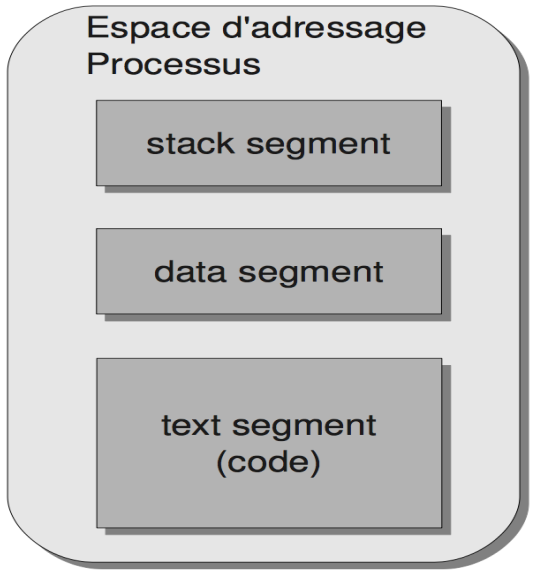


Figure: Espace d'adressage d'un processus

- Le code contenant les instructions du programme (*text segment* en anglais) ;
- Les variables globales et les variables allouées dynamiquement (*data segment* en anglais) ;
- La pile, où les variables locales de ses sous-programmes, ainsi que diverses informations temporaires ayant une durée de vie égale au sous-programme sont stockées (*stack segment* en anglais).

Un processus, dans un cadre multi-threadé, est décomposé en deux parties :

- La première contenant les ressources globales, telles que les instructions du programme et les variables globales. Cette partie correspond au *processus*. Il s'agit des deux premiers points de l'espace d'adressage ;
- La deuxième contenant des informations liées à l'état d'exécution, telles que le *compteur de programme* (aussi appelé *compteur ordinal*) et la *pile d'exécution*. Cette partie correspond à un thread. Il est à noter que chaque thread possède un compteur de programme et une pile. Il s'agit de la partie liée au *thread*.

Contenu

Un *thread* est un fil d'exécution de code, à l'intérieur d'un processus, et qui a la possibilité d'être ordonnancé. Il s'agit d'une version allégée d'un processus. Processus et threads partagent certaines propriétés, mais ne peuvent en aucun cas être interchangés. Tout processus a un thread principal, depuis lequel d'autres threads peuvent être lancés, dans le cas d'une application multi-thread.

Les threads d'un même processus partagent le même espace d'adressage, comme illustré à la figure 2 Toutes les ressources du processus peuvent être accédées par tous les threads, ce qui n'est pas le cas entre deux processus distincts. Toutefois, chaque thread possède son propre compteur programme (PC), un ensemble de registres, un état, et une pile. Les piles sont placées dans l'espace mémoire dédié aux piles, mais chaque thread possède sa propre pile. Les variables globales sont par contre partagées entre les threads.

- 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.

Contenu

- 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.

Contenu

- 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.

Contenu

- 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.

Contenu

- 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.

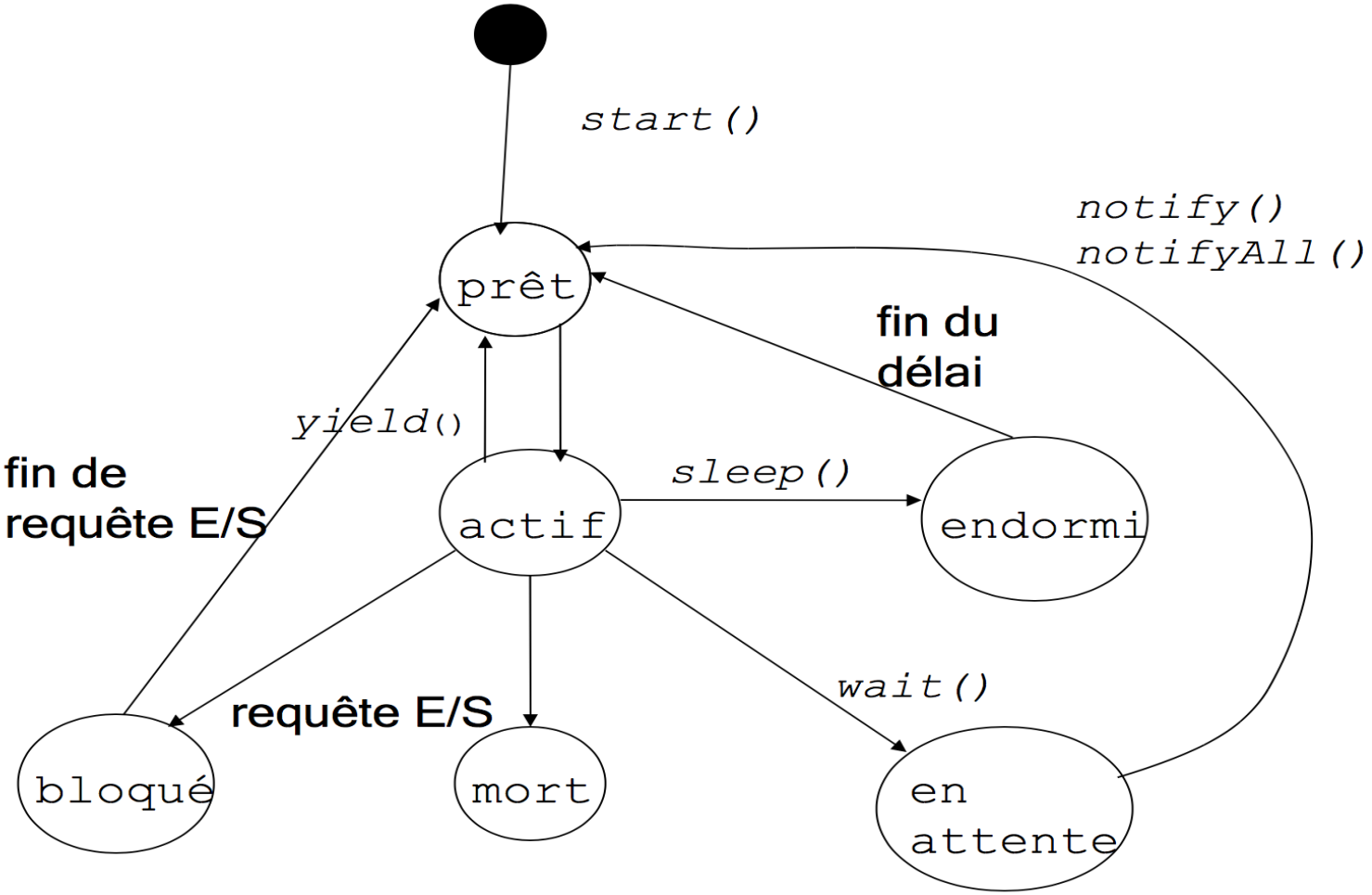
- En fait, il existe plusieurs files d'attente correspondant aux différents niveaux de priorité. Il existe aussi plusieurs politiques d'ordonnancement ;
- En général, la file de plus haute priorité est d'abord explorée et tant qu'elle contient des processus, ceux ci sont privilégiés pour l'attribution 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.

Contenu

- Supposons que chaque quantum ait une durée de 1 ms ;
- A 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.

Contenu

- Ce sont des instances d'une classe dérivée de la classe *Thread* ;
- 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
actif
sleep()
endormi
wait()
requête E/S
bloqué
mort
en
attente

Contenu

- 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.

Contenu

```
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();
}
```

Contenu

```
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 );
    }
}
```

```
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

Contenu

- 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*.

Contenu

```
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 );
    }
}
```

```
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

2222222222
priorité actuelle : 5
nombre de threads : 6
3333333333
1111111111

résultats possibles affichés

- 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

Contenu

- Permettre à un serveur d'accepter plusieurs clients en parallèle est très similaire ;
- Il faut cependant une classe héritée de la classe *Thread* qui servira toutes les opérations propres à un client particulier.

```
import java.net.*;
public class ServeurMultiThread{
  public static final int PORT = 8080;
  public static void main(String[] args)throws IOException{
    // s écoute les connexions entrantes
    ServerSocket s = new ServerSocket(PORT);
    System.out.println( s + " à l'écoute!" );
    try{
     while (true){
       // processus bloqué jusqu'à une demande de connexion
       Socket socket = s.accept();
       try{
         new SertUnClient( socket );
       }
       catch(IOException e){
          socket.close();
       }
       }
     }
      finally{ s.close(); }
    }
}
```

```
import java.net.*;
public class SertUnClient extends Thread
{
  private Socket socket;
  private BufferedReader br;
  private PrintWriter out;
  public SertUnClient( Socket s ) throws IOException
  {
    socket = s;
    InputStreamReader isr =
       new InputStreamReader( socket.getInputStream() );
    br = new BufferedReader(isr);
    OutputStreamWriter osw =
     new OutputStreamWriter( socket.getOutputStream() );
    BufferedWriter bw = new BufferedWriter(osw);
    out = new PrintWriter(bw,true);
    start();
  }
```

```
public void run()
{
   try
   {
      while (true)
      {
        String str = br.readLine();
        if (str.equals( "FIN" )) break;
        System.out.println( "echo : " + str);
        out.println( str );
      }
     System.out.println( "fermeture de la   connexion!" );
   }
   finally
   {
     try{ socket.close(); }
     catch(IOException e)
     {
       System.err.println( "Socket non fermée!" );
     }
   }
}
```

Contenu

- 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;
} }
```

Contenu

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€.

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*

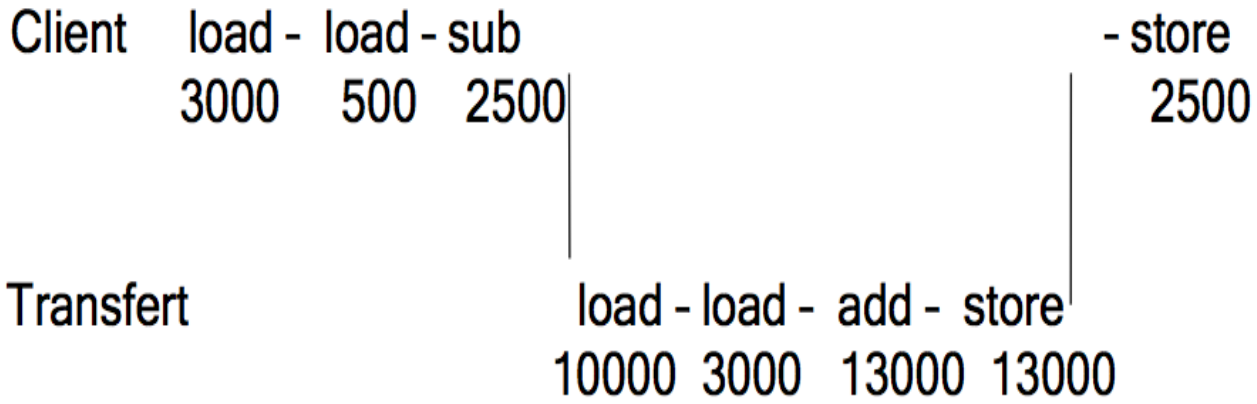
Le "bytecode" correspondant à l'opération *solde = solde-somme* est :

load_1 solde
load_2 somme
sub 1,2
store solde

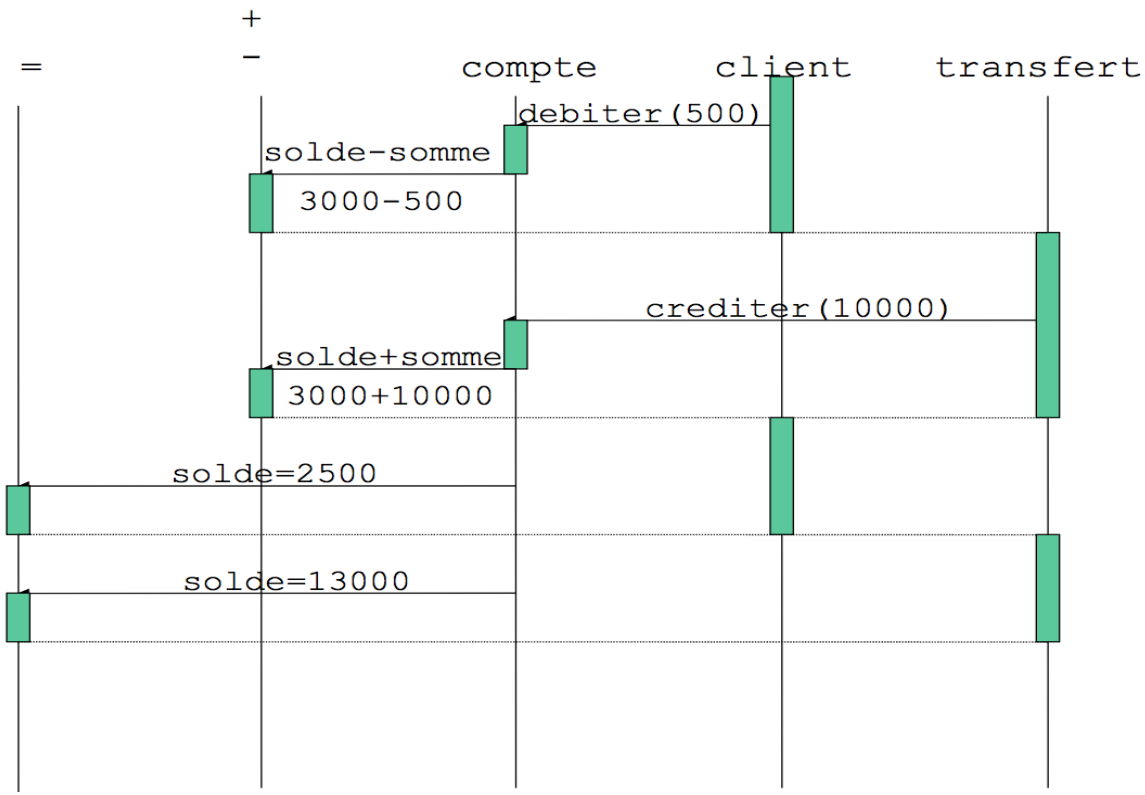
Au niveau des instructions du bytecode, un entrelacement des instructions des différents threads est possible $\Rightarrow$ *nécessité de créer des sections critiques de code java*

Dans l'exemple, les méthodes *crediter* et *debiter* doivent être exécuter sans risque d'être interrompues, en exclusion mutuelle

Contenu



La somme transférée est perdue.


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

Contenu

- Lorsque des threads partagent certaines données de l'application, il y a nécessité de conserver une certaine cohérence sur les données partagées $\Rightarrow$ 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 l' objet obj partagé par *t1* et *t2*
- *t1* tente d'exécuter *b1*, deux cas :
  - le verrou de *obj* est ouvert $\Rightarrow$ *t1* verrouille obj, exécute *b1* et libère le verrou ;
  - le verrou de *obj* est fermé $\Rightarrow$ *t1* est mis en attente jusqu'à l'ouverture du verrou.

Contenu

- 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() {...}*,
  en invoquant *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){...}*

Contenu

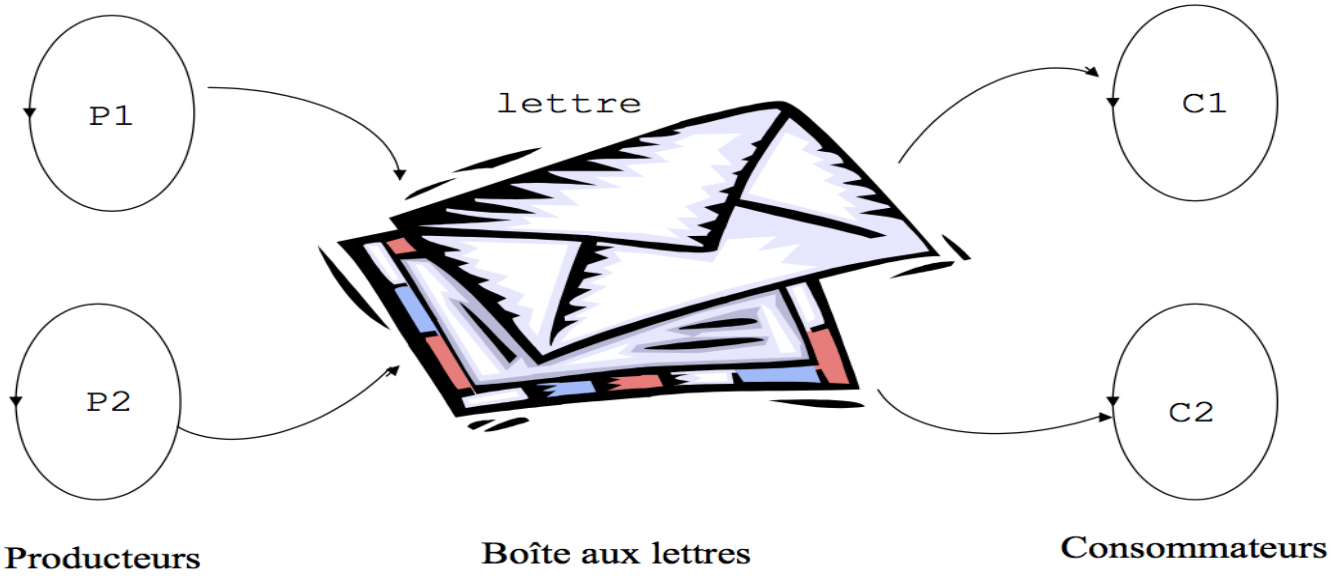
```
public class Compte{
   private double somme;
   public synchronized void crediter( double somme ){
      solde = solde+somme;}
   public synchronized void debiter( double somme ){
      if( solde < somme)
       throw new SoldeInsuffisant();
      else
       solde = solde-somme;
   }
}
```

Contenu

- 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.

Contenu

- 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 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();
    }
}
```

Resultat

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

Contenu

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

    public synchronized String retirer(String destinataire)
    {
     try{
       while( !ok ) wait();// La boîte aux lettres est vide
     }catch( InterruptedException e )
      {
        System.out.println( "interruption" );
        System.exit(1);
      }
     System.out.println(destinataire+" lit "+lettre);
     ok = false;// La boîte aux lettres est de nouveau vidé
     notifyAll();
     return lettre;
```

```
public synchronized void deposer( String lettre )
{
  try
  { while( ok ) wait(); }// on ne peut pas déposer
   // car la boîte aux lettres est pleine
  catch( InterruptedException e )
  {
    System.out.println(" interruption" );
    System.exit(1);
  }
  this.lettre = lettre;
  System.out.println( "dépot de : "+lettre );
  ok = true;// une lettre est de nouveau déposée
  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

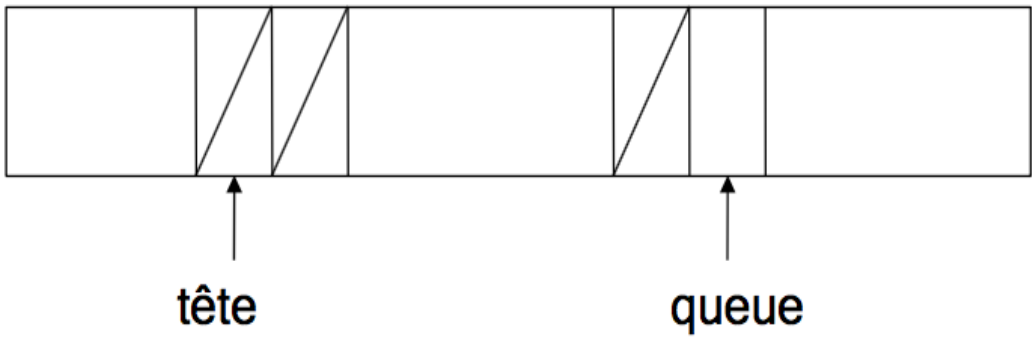
Contenu

- 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()*.

Contenu

- 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.

Le tampon est géré comme une file circulaire.
On ajoute en queue et on retire en tête.



On ne peut pas retirer d'élément lorsque tête==queue
On ne peut pas ajouter d'élément si queue==tête

Contenu

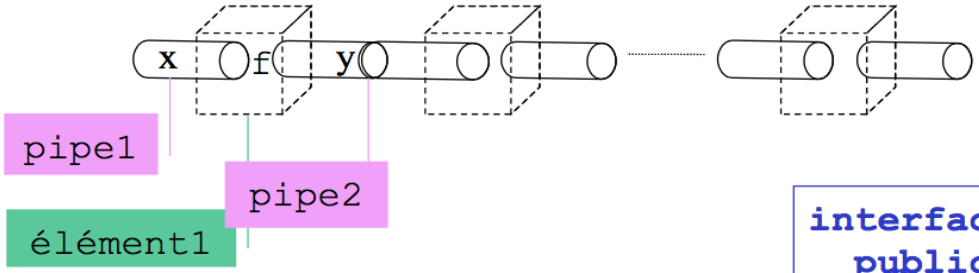
```
public class BoiteAuxLettres {
    private String[] lettres;
    private int tete = 0;private int queue = 0;
    private boolean ecriturePossible = true;
    private boolean lecturePossible = false;
    private static final int TAILLE = 5;

    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;
    }
```

```
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
```

Contenu

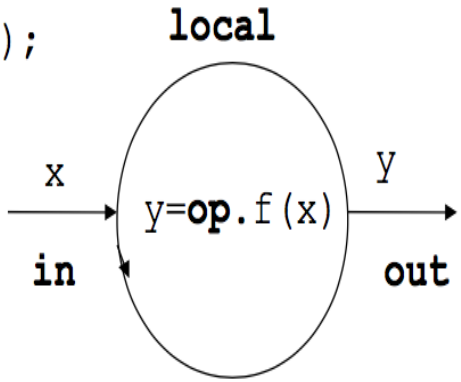



```
interface Operation{
  public int f(int x);
}
```

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

Contenu

```
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);
  }
 }
```



Contenu

```
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; // une lecture a eu lieu
  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;// une écriture a eu lieu
```

Contenu

```
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();
 }}
```

Contenu

```
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;
 }
}
```

Contenu

- Les applets peuvent aussi contenir des threads. Par exemple un thread qui déplace une figure ;
- L'applet doit être active uniquement lorsque la fenêtre apparaît sur l'écran (*start()*) ;
- Lorsque la fenêtre disparaît, le thread doit être interrompu (*interrupt()*) ;
- Lorsque l'applet est détruite, tous les threads doivent aussi être détruits.

```
public class Applette extends Applet implements Runnable
{
  private Thread thread;
  public void init(){//initialisation variables d'instances}
  public void start()
  {
    if ( thread==null )
    {
      thread = new Thread(this);
      thread.start();
    }
  }
  public void stop()
  {
    if ( thread!=null )
    {
      thread.interrupt();
      thread = null;
    }
  }
  public void run(){....;repaint();}
  public void paint( Graphics g ){//dessiner la figure} ..
```