Chapitre II : La gestion des ressources
Notions fondamentales

Eric.Leclercq@u-bourgogne.fr

Département IEM
http://ufrsciencestech.u-bourgogne.fr
http://ludique.u-bourgogne.fr/leclercq

mise-à-jour : Décembre 2016

Plan

La gestion des ressources

Comme nous l'avons vu dans les chapitres précédents, une des premières fonctions d'un SE est la gestion des ressources de l'ordinateur.

Nous allons étudier dans ce chapitre et les suivants les principes de la gestion :

- du processeur
- de la mémoire centrale
- des supports de stockage (disques, bandes etc.)

L'évolution de la notion de processus

- La notion de processus est basée sur celle de programme
- Le terme processus apparu dans les années 60 avec les embryons de système d'exploitation (moniteurs)
  - Préhistoire : *open shop*, *closed shop*
  - Traitement par lots : *batch processing*
  - Apparition des moniteurs permettant de lancer les programmes et de les gérer : *Disk Operating Systems*
  - Développement de la multi-programmation puis de son évolution naturelle vers le temps partagé (*time-sharing*)

Définitions

Définition : (Processus)

Un processus est un programme en exécution

Définition : (SE multitâches)

Un système d'exploitation est dit multitâches s'il permet d'exécuter, apparemment simultanément, plusieurs processus.

- Ce fonctionnement est réalisé en alternant rapidement l'exécution de différents processus c'est-à-dire en effectuant un multiplexage temporel du processeur (chaque processus s'exécute pendant une fraction de seconde)
- L'exécution des processus est entremêlée
- Par conséquent, il ne s'agit pas réellement d'un traitement simultané sauf si la machine à plusieurs processeurs ou cœurs

La commutation de contexte

Définition : (Commutation de contexte)

Le passage de l'exécution d'un processus à un autre est appelé la **commutation de contexte**.

Lors d'une commutation de contexte, le système doit sauver l'état du processus en cours (le compteur ordinal (*Instruction Pointer*), les valeurs des registres, etc.) pour les restaurer plus tard.

Deux stratégies peuvent être envisagées pour la commutation de contexte :

- le mode coopératif
- le mode préemptif

La commutation de contexte

Le multitâche coopératif : chaque processus doit explicitement permettre à un autre d'accéder au processeur

- ce mécanisme a été utilisé dans les SE Microsoft Windows jusqu'à Windows 3.11 et dans Mac OS jusqu'à Mac OS 9
- cette approche simplifie l'architecture du SE mais présente plusieurs inconvénients :
  - si un des processus ne passe pas la main à un autre processus, le système entier peut être bloqué ;
  - le partage des ressources (CPU, mémoire, disque, etc.) peut être inefficace (*IO bound*)

La commutation de contexte

Le multitâche préemptif : le processeur signale au système d'exploitation que le processus en cours d'exécution doit être mis en sommeil pour permettre l'exécution d'un autre processus.

À la différence du multitâche coopératif, la commutation de contexte est transparente.

D'une façon générale la stratégie de commutation choisie à un impact énorme sur l'utilisation des ressources.

Une question subsiste : comment peut-on réaliser la commutation de contexte dans le mode préemptif ?

- Understanding the Linux Kernel, Third Edition, Daniel P. Bovet, Marco Cesati, Oreilly, November 2005, pp. 942
- Linux System Programming, Robert Love, Oreilly September 2007, pp. 388

De nombreux chapitres disponibles en *preview* sur le site `http://www.oreilly.com`

Gestion des ressources en environnement multitâche

Problématique d'une façon générale il s'agit d'étudier comment allouer des ressources aux processus, plus précisement :

- comment identifier les processus ?
- comment partager le CPU entre plusieurs processus ?
- comment restaurer l'état de chaque processus ?
- comment allouer et gérer la mémoire attribuée aux processus ?
- comment gérer les E/S dans un environnement multitâches ?

Très bon site de David Decotigny et Thomas Petazzoni :
`http://sos.enix.org/fr/PagePrincipale`

Multitâche / multi-utilisateurs

- Avec la multi-programmation, plusieurs programmes peuvent être traités en même temps : c-à-d en cours d'exécution et non terminés
- Ce procédé permet l'emploi simultané de la machine par plusieurs utilisateurs
- Grâce à ce mécanisme chaque utilisateur a l'impression d'avoir la machine pour lui seul
- La machine traite alors des processus qui sont chacun la propriété d'utilisateur (humain ou virtuel)

Classification des processus

- Un certain nombre de processus existent aussi longtemps que l'ordinateur est en fonctionnement

Exemple :

la gestion des connexions utilisateur, gestion des disques, le processus `init`

- A contrario, d'autres processus ont une durée de vie limitée

Exemple :

le processus de la commande `date` *n'existe que le temps de la commande*

- Certain processus ont un accès privilégié au SE : propriété de l'administrateur ou d'utilisateurs privilégiés
- Certain processus sont sous le contrôle des utilisateurs (*user-mode*) , d'autre sous le contrôle du SE (exécution en mode *kernel*)

Filiation des processus

- De façon générale le SE alloue un processus pour l'interface utilisateur (un shell par exemple) puis d'autres processus (fils du premier pour chaque commande utilisateur ou pour chaque programme lancé à partir du shell
- Ainsi, il se crée une hiérarchie de processus (père fils) prenant la forme d'un arbre

```
jaromil ~ # pstree
init-+-adsl-connect---pppd---pppoe
     |-6*[agetty]
     |-bdflush
     |-cron
     |-cupsd
     |-devfsd
     |-gconfd-2
     |-gdict-applet
     |-gdm---gdm-+-X
     |           `-gnome-session---ssh-agent
     |-gnome-netstatus
     |-gnome-terminal-+-bash---evolution---evolution---8*[evolution]
     |                |-bash
     |                |-bash---ssh
     |                |-gnome-pty-helpe
     |                `-gnome-terminal---gnome-terminal
     |-gweather-applet---gweather-applet---3*[gweather-applet]
```

Filiation des processus

- Le système d'exploitation d'une machine personnelle gère une centaine de processus, celui d'un serveur plusieurs centaines
- Il existe une limite maximum fixée dans le noyau

Exemple :

pour un SE Linux le fichier concerné est `include/linux/tasks.h`, la variable `NR_TASKS` fixe la limite à 4096. Dans les noyaux 2.4 la limite est une fonction de la taille mémoire disponible.

Les modes de lancement des processus

Un processus donné à trois façons pour créer un processus fils :

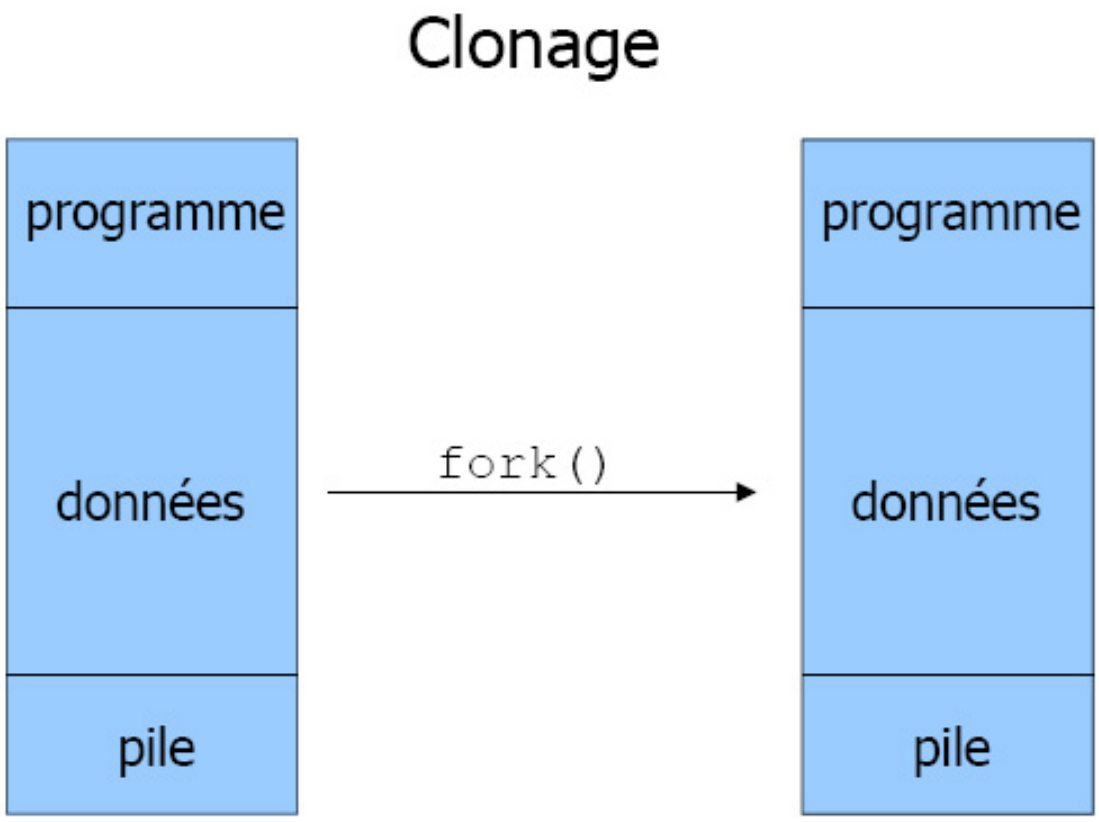
- clonage : processus parallèle identique au père mais indépendant ;
- recouvrement par le fils ;
- mise en route d'un processus fils ne comportant que certaines portions du processus parent.

Ces différentes stratégies permettent de fournir une manière efficace pour organiser l'exécution des applications dans le SE.

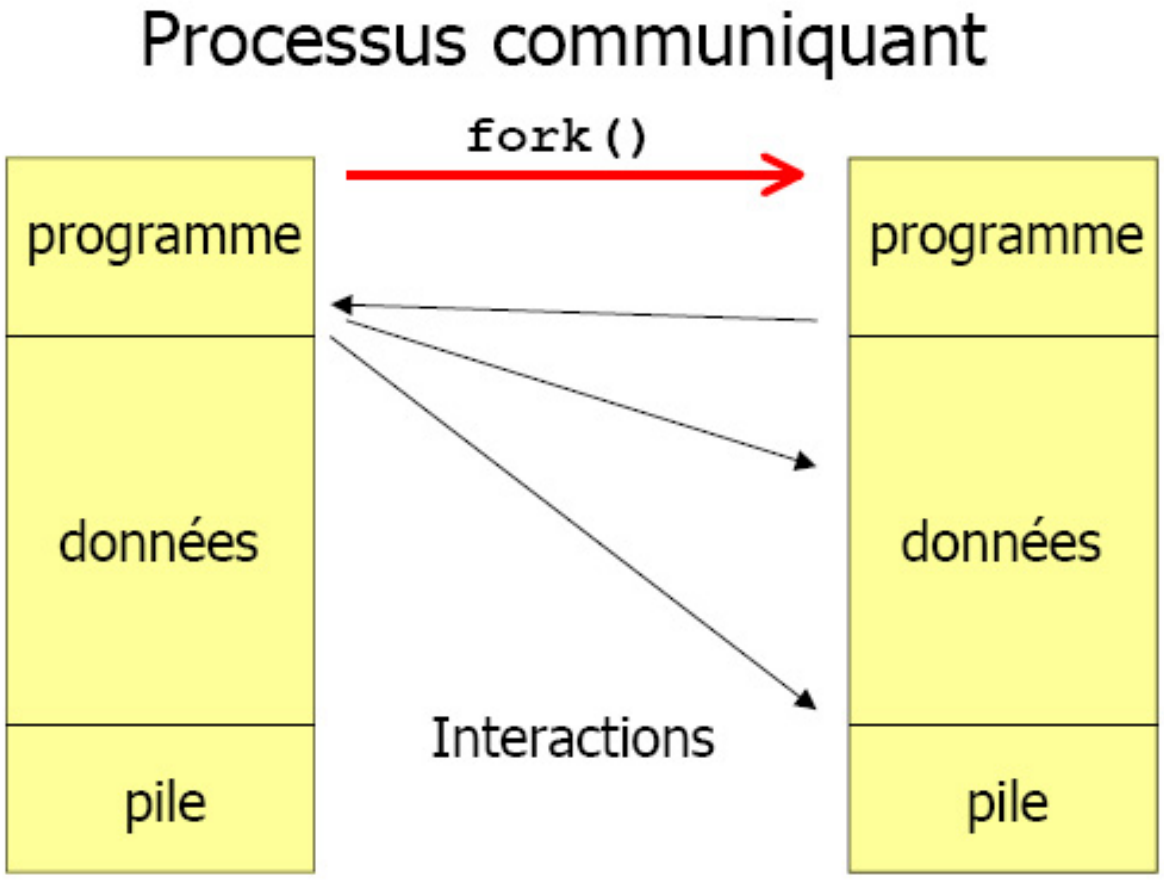
La majorité des SE multi-tâches propose les trois modes de création

Comment sont gérées les interactions père fils et attente de terminaison ?

Clonage


Clonage
programme
données
pile
fork()
programme
données
pile

Clonage et communication : stratégies



Exemple de création par clonage

L'appel système `fork()` crée un nouveau processus :

- après l'exécution le pointeur d'instruction du fils et du père sont positionnés sur l'instruction suivante
- `fork()` retourne la valeur 0 au fils
- `fork()` retourne un entier non nul (identifiant du processus fils) au père

Exemple :

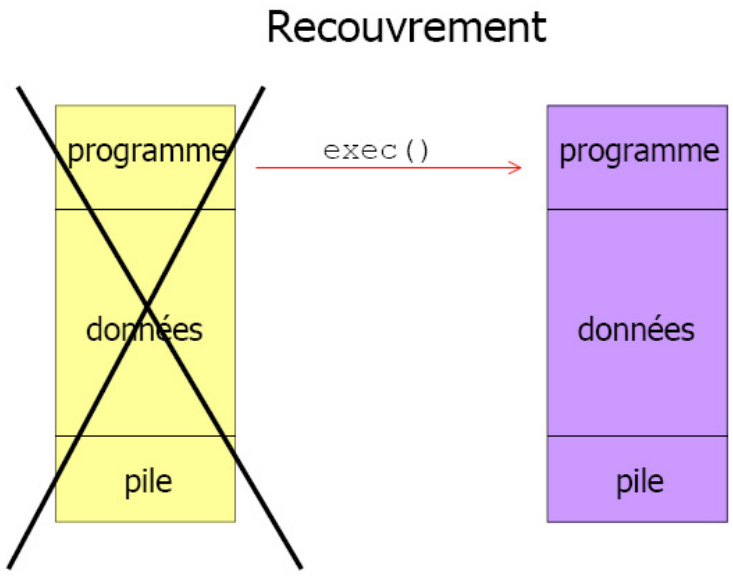
```
id-fils=fork()
if (id-fils==0)
 traitements relatif au fils
else
 traitements relatif au père
```

Exemple de création par clonage

Exemple :

```
...
pid = fork();
switch(pid) {
 case -1:   /* erreur dans fork() */
   fprintf(stderr,"error %d in fork: %s\n",errno,sys_errlist[errno]);
   exit(errno);
 case 0:    /* on est dans le fils */
   fprintf(stdout,"Dans le fils, pid = %d\n",getpid());
   sleep(20);
   break;
 default:   /* on est dans le pere */
   fprintf(stdout,"Dans le pere, pid = %d\n",getpid());
   sleep(20);
```

Exemple de création par recouvrement



La création par recouvrement se fait au moyen d'une opération de la famille exec

Exemple :

```
...
 printf("Exemple de recouvrement\n");
 execve("/bin/ls", argv, envp);
 printf("Ne s'affiche pas");
...
```

Lancement des processus

Ils existe différent types de processus :

- les démons (*daemon*) sont des processus crées par le système hors contrôle des utilisateur. Plus généralement dans les SE Unix, les démons
  - peuvent être relancés automatiquement si ils sont tués
  - avoir comme parent direct le processus *init*
- les zombies : processus sans filiation directe, ne pouvant pas être tués
- les processus utilisateurs, par exemple : applications, commandes, processus de communication crée à partir d'un démon

Note :

À l'origine le terme daemon signifiait Disk And Execution MONitor c-à-d un mini système d'exploitation pour contrôler les disques et l'exécution des programmes.

Les processus légers (*Thread*)

Certains SE proposent en plus de la notion de processus, un mécanisme de processus légers appelés *thread* ou fil d'exécution. Ce mécanisme permet :

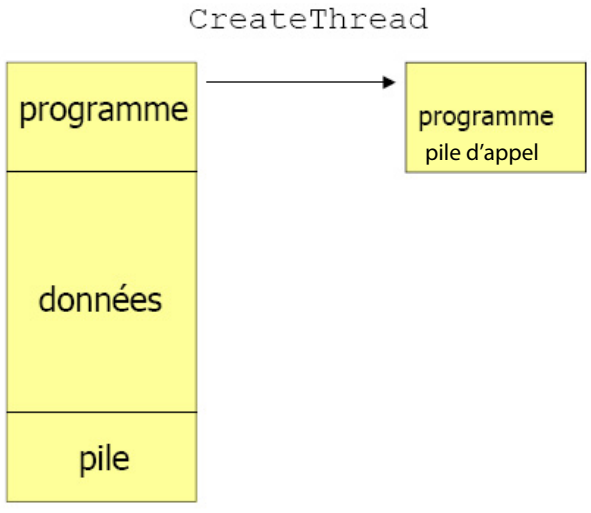
- une création rapide des processus fils
- une communication simplifiée entre un processus père et ses enfants

Le processus père et le ou les threads fils partagent la zone de données du programme.

Note :

La commande `ps -efL` permet de visualiser les thread sous Linux. qps `http://qps.kldp.net/`, top sont des commandes plus synthétiques.

Les processus légers (*Thread*)



Les threads existent dans de nombreux SE : la famille UNIX Windows NT et 2000 WIN 9x et Me

Dans cette dernière famille (9x, Me), la notion de processus n'existe pas : les programmes lancés depuis l'interface graphique sont des threads.

Ainsi le système peut être rendu instable si un programme endommage la zone de mémoire commune.

Les modèles de Thread

Les threads font partie du processus donc dans la partie utilisateur du SE, elle doivent être mises en correspondance avec des threads du noyau :

- N:1 (User-level threading)
  - GNU Thread
- 1:1 (Kernel-level threading)
  - Native POSIX Thread Library (NPTL) Linux, est une implémentation des POSIX Threads (pthreads) standards
  - Light Weight Kernel Threads (LWKT) famille BSDs
  - Microsoft Windows noyaux NT et suivants
- M:N (Hybrid threading)
  - Windows Seven
  - Version Solaris antérieures à 8

Les modèles de Thread et langages de programmation

Les modèles de thread sont implantés par les bibliothèques dans les langages de programmation avec d'éventuelles abstractions:

- Dans la langages impératifs : C, C++, Java, etc.
- Dans les langages déclaratifs :
  - Scala, `http://www.scala-lang.org/`
  - Erlang, `http://www.erlang.org/`
  - Clojure, `http://clojure.org/`

Implémentation des Thread dans Java

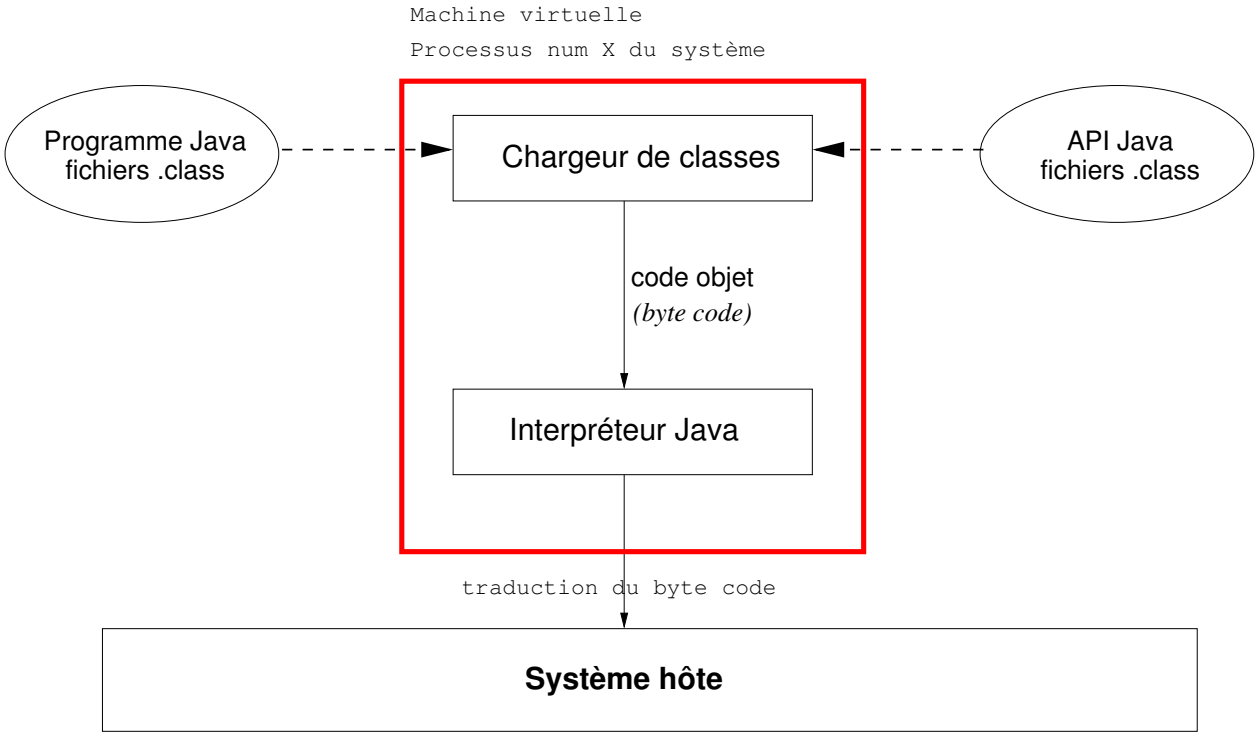
La plateforme Java propose une machine virtuelle (MV) Java qui est un environnement d?exécution permettant d?exécuter des programmes Java indépendamment du processeur hôte et du système d'exploitation.

Une machine virtuelle réalise une abstraction des machines réelles :

- elle propose un interpréteur de byte-code : langage machine/instructions d'un processeur virtuel traduites dans en instructions pour le processeur réel
- elle fournit un mécanisme d'interaction avec le système d'exploitation hôte (Linux, Windows, MAC OS X etc.)
- elle se comporte comme un système d'exploitation : gestion de la mémoire dans la MV, gestion des processus légers, gestion des E/S

Implémentation des Thread dans Java

La machine virtuelle est lancée au moyen de la commande `java` et elle devient un processus du système d'exploitation



Les fichiers `.class` sont les mêmes quelquesoit le système d'exploitation : inutile de recompiler un application si on change de SE ou de matériel.

Les Thread dans Java

Le langage Java propose deux modes de création des threads via l'héritage ou l'implémentation d'interface (héritage de fonctionnalité) :

```
1  class Worker1 extends Thread{
2   public void run() {
3     System.out.println("I␣Am␣a␣Worker␣Thread");
4     }
5  }
6
7  class Worker2 implements Runnable{
8   public void run() {
9     System.out.println("I␣Am␣a␣Worker␣Thread");
10    }
11 }
12
13 public class First{
14  public static void main(String args[]) {
15    // une premiere forme d'utilisation
16    Worker1 thr1 = new Worker1();
17    thr1.start();
18    System.out.println("I␣Am␣The␣Main␣Thread");
19    // une seconde forme d'utilisation
20    Runnable runner = new Worker2();
21    Thread thrd = new Thread(runner);
22    thrd.start();
23    }
24 }
```

Interaction de la MV avec le SE

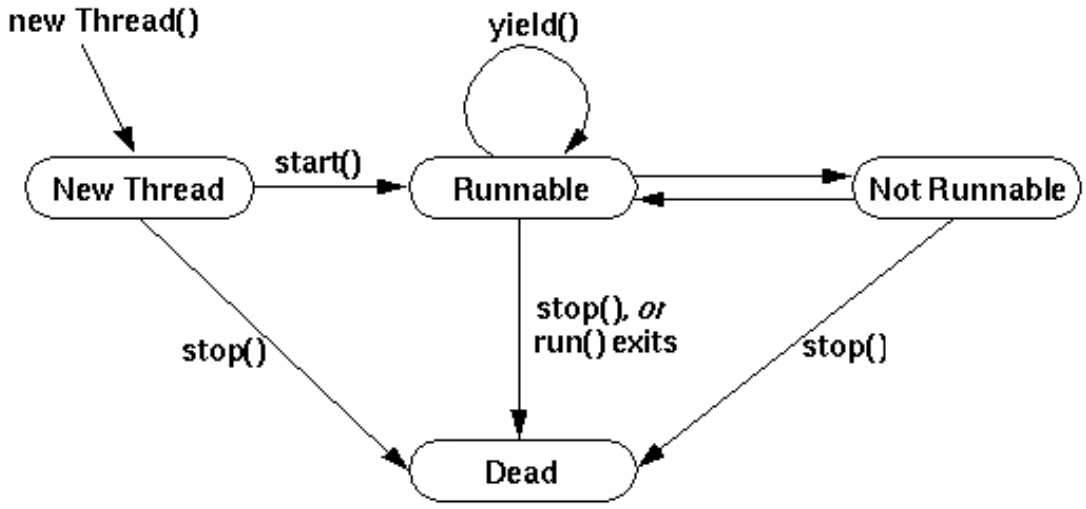
```
1  import java.io.*;
2  public class OSProcess {
3   public static void main(String[] args) throws IOException {
4    if (args.length != 1) {
5     System.err.println("Usage: java OSProcess <command>");
6     System.exit(0);
7    }
8
9   // args[0] correspond au processus à créer (commande ou programme)
10  ProcessBuilder pb = new ProcessBuilder(args[0]);
11  Process proc = pb.start();
12
13  // crée des flux pour les E/S
14  InputStream is = proc.getInputStream();
15  InputStreamReader isr = new InputStreamReader(is);
16  BufferedReader br = new BufferedReader(isr);
17  String line;
18  while ( (line = br.readLine()) != null)
19  System.out.println(line);
20  br.close();
21  }
22 }
```

Le cycle de vie des processus

Un processus peut avoir trois états principaux :

- en train d'être exécuté : on dit qu'il est actif
- en attente d'être exécuté : il est prêt (toutes les ressources dont il a besoin sont disponibles)
- bloqué : il est en attente de ressources ou d'un signal provenant d'un périphérique
- mort : il n'existe plus et la place qu'il occupe dans le système va être libérée

Le cycle de vie des Thread Java



Contrôle des threads

Pour gérer l'exécution des threads les méthodes proposées sont :

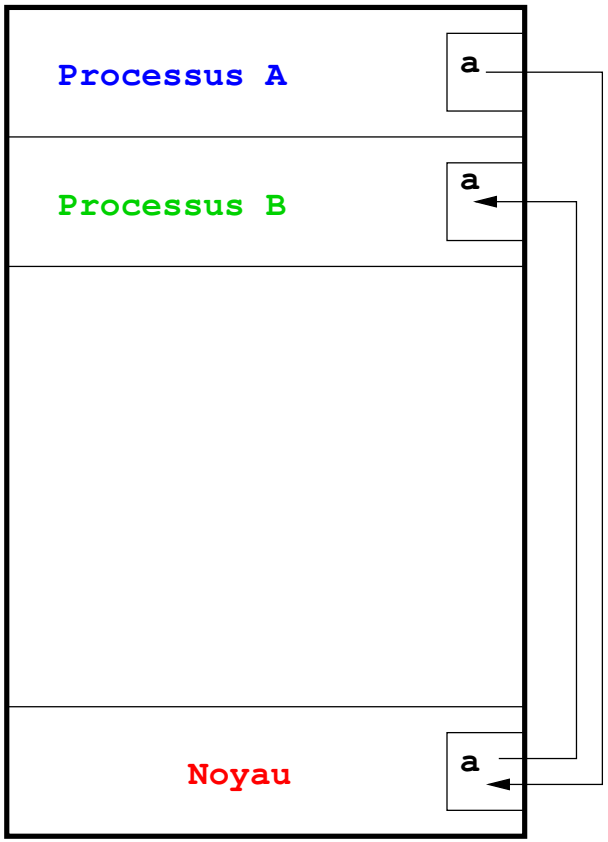
- `public void start()` : permet de démarrer un thread, la méthode start, lance un nouveau thread dans le système dont le code à exécuter démarre par le run. Si vous invoquez la méthode `run` le code s'exécute bien, mais aucun nouveau thread n'est lancé dans le système.
- `public void suspend()` : permet d'arrêter temporairement un thread en cours d'exécution.
- `public void resume()` : permet de relancer l'exécution d'un thread, au préalable mis en pause via `suspend`.
- `public void stop()` : permet de stopper, de manière définitive, l'exécution du thread. Une autre solution pour stopper un thread consiste à simplement sortir de la méthode run.

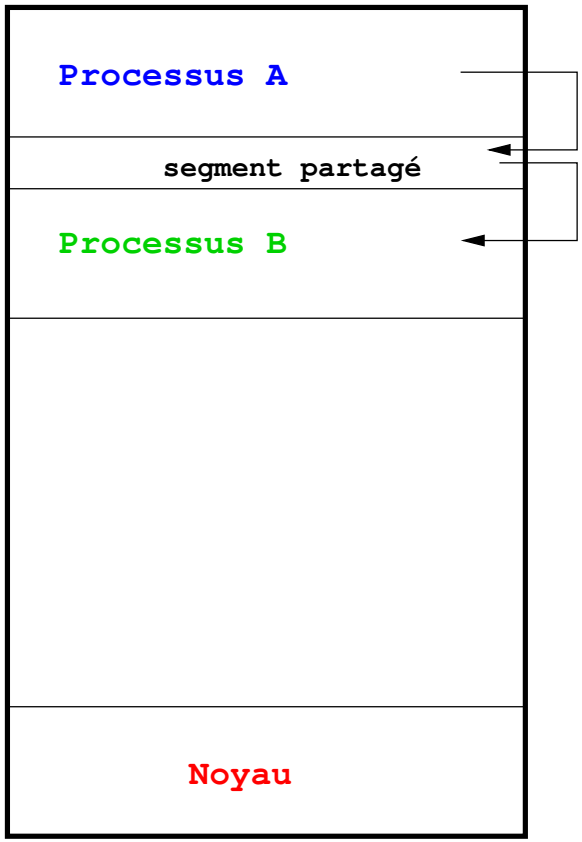
Les différents modes de communication

Le terme IPC (*Inter Process Communication*) recouvre les modes de communication entre processus à l'intérieur du SE ou entre plusieurs SE sur des machines différentes :

- communication par message (*Message Passing*)
- mémoire partagé et buffering (*Shared Memory*)
- appel de procédures / méthodes (*Remote Procedure Call*)
- socket pour les systèmes distribués (chapitre 7) communiquant via un réseau

Les deux modèles principaux de communication





Passage / échange de messages

Deux fonctions de bases sont proposées pour la communication entre processus :

- `send(message)`
- `receive(message)`

Si les processus P et Q souhaitent s'envoyer des messages un lien de communication est nécessaire :

- Lien direct : `send(P, message)` et `receive(Q,message)`
- Lien indirect avec boîte aux lettres : `send(A, message)` et `receive(A,message)`
- Lien synchrone : envoi bloquant ou non, réception bloquante ou non
- Lien bufferisé avec file d'attente

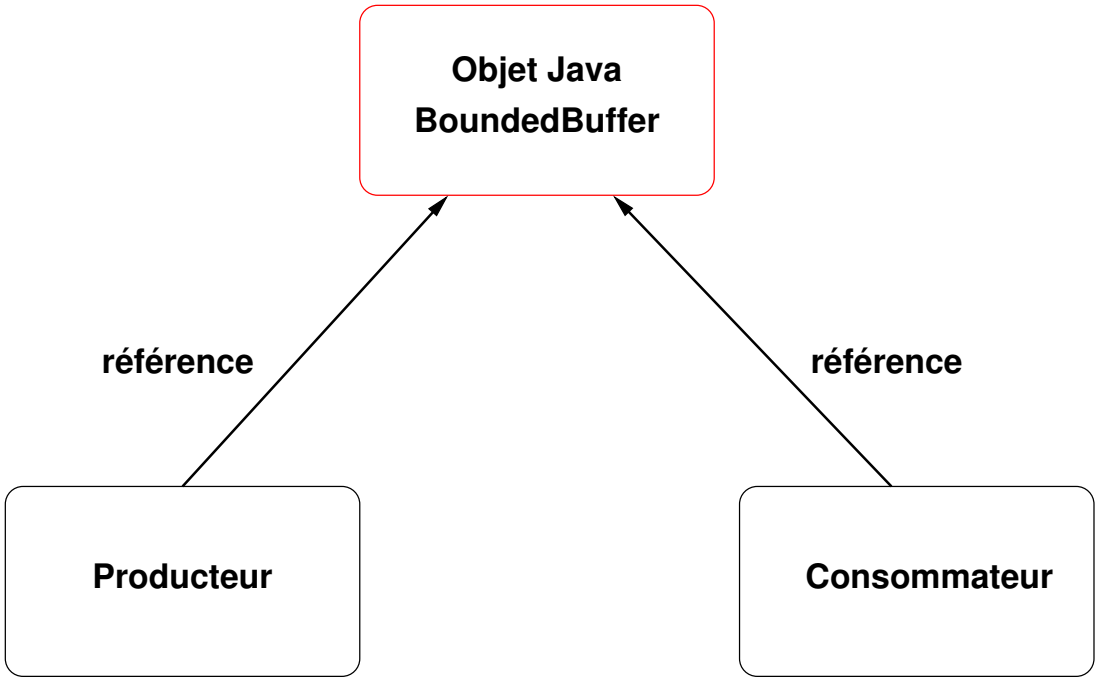
Mémoire partagée et *buffer*

- Une zone de mémoire est déclarée comme partagée par un processus et peut être utilisée ensuite comme un moyen de communication entre plusieurs processus.
- Les paradigme de coopération est le suivant : un processus producteur écrit des informations dans le buffer qui sont consommée par un processeus consommateur.
- Le buffer peut être limité (*bounded-buffer*) ou s'étendre en fonction des capacité mémoire du système (*unbounded-buffer*)

Note :

Lors de l'utilisation de threads, la zone de mémoire utilisée par le parent et partagée automatique avec les thread crées.

Mémoire partagée et buffer en Java




```
1  public interface Buffer{
2
3   // les producteurs utiliseront cette méthode
4   public abstract void insert(Object item);
5
6   // les consommateurs utiliseront cette méthode
7   public abstract Object Remove();
8  }
```

Appel de procédure (RPC)

Ce mécanisme est essentiellement utilisé pour des communications au travers d'un réseau.

On parle alors de RPC (*Remote Procedure Call*).

Un système similaire est proposé pour les langages orientés objet (Java RMI pour la langage Java).

Hors du domaine couvert par le cours de L2.

La notion de processus concurrents

Définition : (Ressource)

La notion de ressource dans les SE désigne tout élément qui est utile au déroulement d'un processus

Une ressource peut être :

- Physique (si on se place du point de vue du système d'exploitation) : processeur, mémoire, périphérique
- Logique (si on se place du point de vue du programmeur) : fichier, variable, objet

La notion de processus concurrents

Plusieurs processus peuvent avoir besoin des mêmes ressources.

Définition :

une ressource est dite partageable si elle peut être utilisée en même temps à plusieurs processus. Une ressource est dite non partageable si elle doit être à utilisée exclusivement par un seul processus.

Dans ce cas on dit que les processus doivent être exclusion mutuelle pour l'accès à la ressource.

La notion verrouillage

Lorsqu'une ressource est partagée, si plusieurs processus accèdent à la ressource, des incohérences peuvent en résulter : **il faut un arbitre**

Exemple :

des guichets automatiques de retrait d'argent : simulation du comportement

Comment éviter les situation de compétition sur des ressources non partageable ?

- interdire l'accès partagé à la ressource : exclusion mutuelle
- interdire l'exécution simultanée du codé accédant à la ressource : section critique

La notion verrouillage

Pour résoudre ce problème on utilise la notion de verrou (*lock*) deux opérations sont proposées :

- `Verouiller(v)` permet à un processus d'acquérir un verrou, s'il n'est pas disponible, le processus est bloqué en attente du verrou
- `Déverouiller(v)` permet de libérer un verrou

En Java depuis la version 5 sont disponibles plusieurs types de verrous dans le package `java.util.concurrent` :

- l'interface lock
- ReentrantLock
- ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock

Les sémaphores

Les sémaphores (Dijkstra 1965) proposent un mécanisme de contrôle de la concurrence sous la forme d'un distributeur de jeton (n jetons).

Deux opérations sont définies :

- P(s) pour obtenir un jeton : l'opération P est mise en attente jusqu'à ce qu'une ressource soit disponible
- V(s) pour restituer un jeton :

P et V signifient en néerlandais *Proberen* (tester) et *Verhogen* (incrémenter).

Note :

Les opérations P et V doivent être indivisibles : le SE leur garanti une exécution atomique (pas de commutation de contexte).

Si il y a un seul jeton alors le sémaphore est un verrou.

Les moniteurs

Hoare (1974) à proposé une solution de plus haut niveau basée sur le concept de moniteur.

- il s'agit d'une construction spécifique au langage de programmation ;
- un seul processus à la fois peut exécuter une fonction du moniteur ;
- les moniteurs sont implantés dans le langage Java via la notion de méthodes définies avec le modificateur `synchronized`

Les classes utiles de l'API Java

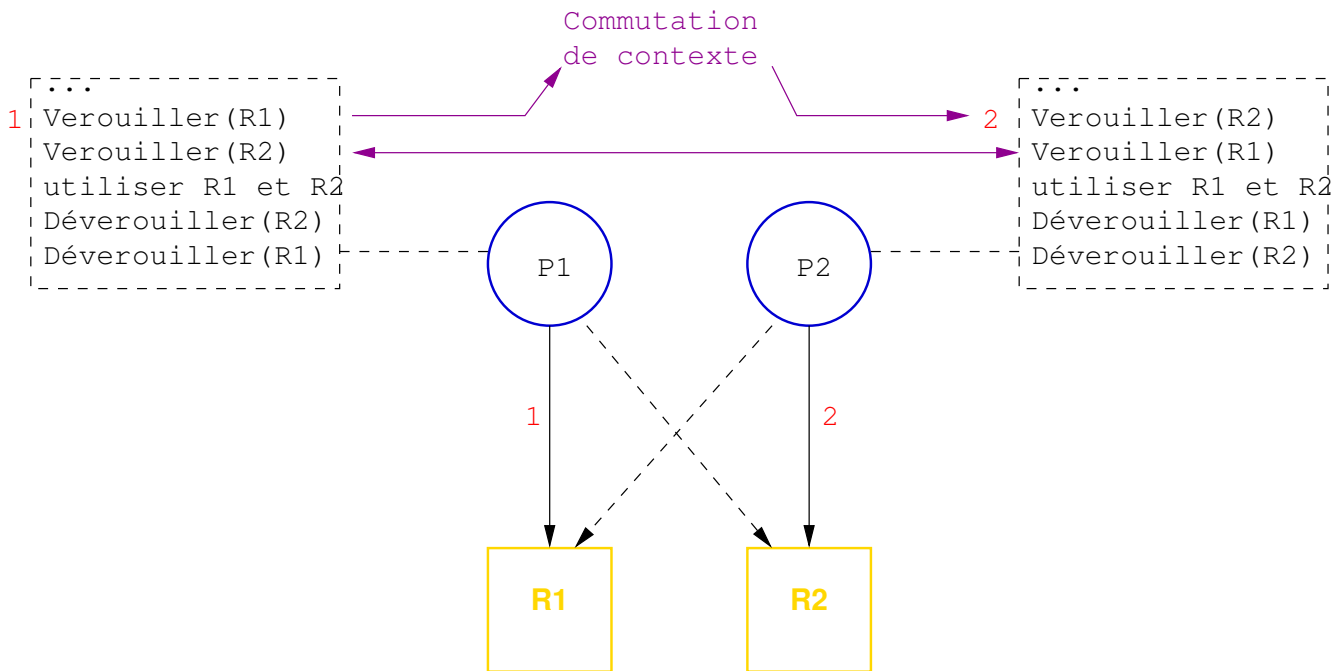
Depuis la version 5 Java propose des fonctionnalités de haut niveau pour compléter les primitives de gestion de la concurrence. La plupart des fonctionnalités sont dans les packages de `java.util.concurrent`.

- Les `Lock` objecs que nous avons déjà évoqué ;
- Les `Executors` proposent des méthodes pour lancer et gérer les thread dont les thread pool ;
- Les concurrent collections pour gérer de grandes collection de données sans avoir à traiter la concurrence ;
- Atomic variables permettent de minimise l'utilisation des primitives de concurrence ;
- `ThreadLocalRandom` (JDK 7) pour la génération de nombres aléatoire (peudo) en utilisant plusieurs thread .

De nouvelles structures de données supportant la concurrence sont aussi proposées dans Java Collections Framework.

Problème des verrous mortels

Le problème des verrous mortels traduit une situation d'interblocage causé par l'accès à une ressource non partageable. On parle également d'étreinte fatale.



Solutions au problème des verrous mortels

Il n'existe pas de solution réellement satisfaisante :

- Politique de l'autruche
- Prévention : plusieurs algorithmes existent dont celui du banquier très connu
- Détection puis guérison : implémentable avec un thread de contrôle (watch-dog)

Synchronisation

La synchronisation des processus est un problème plus large que celui de la concurrence :

- concerne les méthodes mises en œuvre afin que les processus coordonnent leurs activités ;
- attente de la réalisation d'un calcul (fork/join) ;
- utilisation de ressources (modèles Lecteurs/Rédacteurs et Producteurs/consommateurs, etc.) ;
- file d'attente de traitements ;
- barrière de synchronisation etc.

Cette partie est abordée en TD et en TP au travers de plusieurs exemples.

Primitives pour la synchronisation

Java offre deux primitives essentielles (`wait` et `notify`) pour assurer la synchronisation des processus en s'appuyant sur la notion de moniteur (`synchronized`).

- avec la méthode `sleep()` il est possible de mettre en attente un thread pour une durée déterminée, mais cette solution de synchronisation n'est dans la majorité des cas pas correcte.
- `synchronized()` permet de définir une section critique à différents niveaux de granularité :
  - méthode `public synchronized void m(){...}`
  - objet `syncrhonized(o){...}`
  - bloc (d'une méthode d'un objet) `syncrhonized(this){...}`

  Extension de `synchronized` à des éléments statiques et aux classes

Primitives pour la synchronisation

- Tout objet Java possède un verrou interne manipulé par la primitive `synchronized`.
- Tout objet Java possède la liste des threads mis en attentes à la suite d'un appel à la méthode `wait()` : c'est le `wait-set`.
- `wait()` ne peut être appelée que si on à un accès exclusif à un objet, dans ce cas le thread est mis en attente et enregistré dans le `wait-set` associé à l'objet.
- `notify()` et `notifyAll()` ont pour effet de réveiller les threads.

STM : software transactional memory

Le principe des la mémoire transactionnelle logicielle, reprend la notion de transaction des SGBD pour contrôler les accès à la mémoire partagée.

Définition : (transaction de type STM)

Une transaction dans ce contexte est une portion de code qui effectue des lectures écritures sur la mémoire partagée

- les lectures ou écritures se déroulent en une unité de temps
- les états intermédiaires ne sont pas visibles des autres transactions
- peu produire des inversion de priorité et ne résout pas les problèmes de deadlock

Threads et langages fonctionnels

Le modèle de processus et d'état partagé induit des problèmes :

- de concurrence
- de synchronisation
- de verrou mortels
- de famine

Sans partage d'état ces problèmes n'existent plus.
Depuis quelques années les langages fonctionnels pour les applications distribuées et multi-thread se développent :

- Erlang origine Ericsson (utiliser dans les réseaux sociaux)
- Scala origine EPFL (objet et fonctionnel) : fonctionne sur une machine virtuelle Java
- Clojure gagne l'intérêt des développeurs depuis 2 ans

Paradigme d'acteur

Definition

Un acteur (*actor*) est une abstraction logicielle qui implémente un protocole de passage de message permettant de gérer la concurrence.

- Les acteurs n'ont pas d'état partagé, ils communiquent par envoi et réception de messages.
- Ce concept fournit un modèle de concurrence plus simple que le modèle à état (ou mémoire) partagé.
- Évite par conséquent les deadlocks, live locks, thread starvation etc.

L'implémentation dans Erlang à permis de traiter des système massivement concurrents avec plusieurs centaines de milliers d'acteurs.

Paradigme d'acteur

Les acteurs ont les caractéristiques suivants :

- chaque acteur possède une boite aux lettres ;
- un peut recevoir des messages et les filtrer ;
- en fonction du message l'acteur à libre choix de le traiter et de réagir par une action ;
- les acteurs proposent leur services sous la forme de *trait*.

Exemple

Un trait avec un méthode qui réagit au message `status`. Le trait doit être spécialisé pour pouvoir être utilisé par un objet.

```
1    // servers
2    trait Server {
3     def status = println("current␣server:␣" + this)
4    }
5   class ServerOne extends Server
```

Utilisation de la console de Scala, instanciation d'un objet de la classe `ServerOne class`

```
1  $ scala -cp .
2
3  scala> val server = new ServerOne
4  server: ServerOne = ServerOne@7be75d
5
6  scala> server status
7  current server: ServerOne@7be75d
```

Exemple : mise en place des événements

Principes :

- Définir des messages auxquels l'objet répond
- Scala propose des *case classes* similaire aux classes avec quelques ajouts :
  - méthodes pour effectuer du pattern matching
  - pas d'utilisation de new, le compilateur génère les méthodes de base (getter, setter, constructeurs)
  - pas d'égalité de référence mas une égalité de contenu

Deux messages différents : `Status` et `HotSwap`.

```
1 // triggers the status method
2 case object Status
3 // triggers hotswap - carries the new server to be hotswapped to
4 case class HotSwap(s: Server)
```

Exemple : définition de l'acteur

```
1  class ServerActor extends Actor {
2    def act = {
3      println("starting␣server␣actor...")
4      loop(new ServerOne)
5    }
6
7    def loop(server: Server) {
8      react {
9        case Status =>
10         server.status
11         loop(server)
12
13       case HotSwap(newServer) =>
14         println("hot␣swapping␣code...")
15         loop(newServer)
16
17       case _ => loop(server)
18     }
19   }
20 }
```

Exemple : instanciation de l'acteur

```
1  // actor companion object
2  object ServerActor {
3    val actor = {
4      val a = new ServerActor
5      a.start; a
6    }
7  }
```

Exemple : utilisation de l'acteur

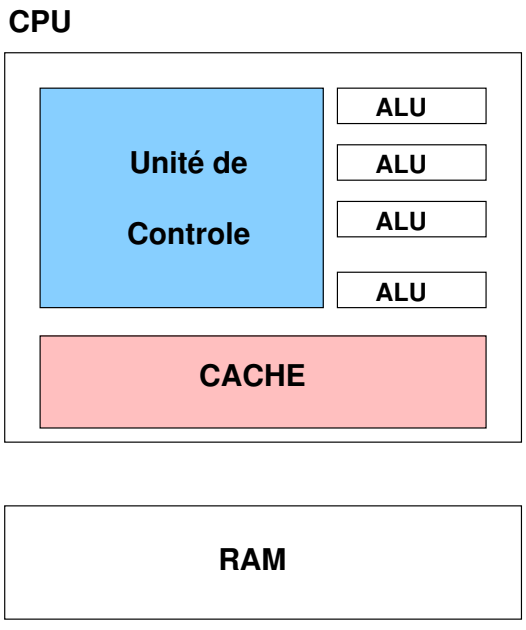
```
$ scala -cp .

scala> import hotswap._
import hotswap._

scala> val actor = ServerActor.actor
starting actor...
actor: hotswap.ServerActor = hotswap.ServerActor@528ed7

scala> actor ! Status
current server: hotswap.ServerOne@226445

scala> class ServerTwo extends Server {
     | override def status = println("hotswapped server: " + this)
     | }
defined class ServerTwo

scala> actor ! HotSwap(new ServerTwo)
hot swapping code...

scala> actor ! Status
hotswapped server: line5$object$$iw$$iw$$iw$ServerTwo@b556
```

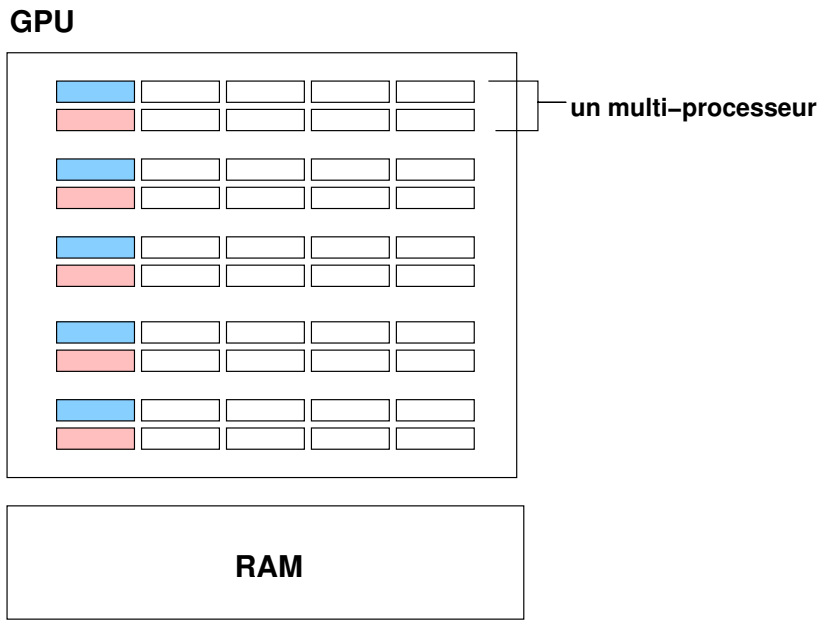
Le modèle de programmation CUDA

CUDA (*Computer Unified Device Architecture*) est une architecture logicielle de traitement parallèle développée par NVIDIA.

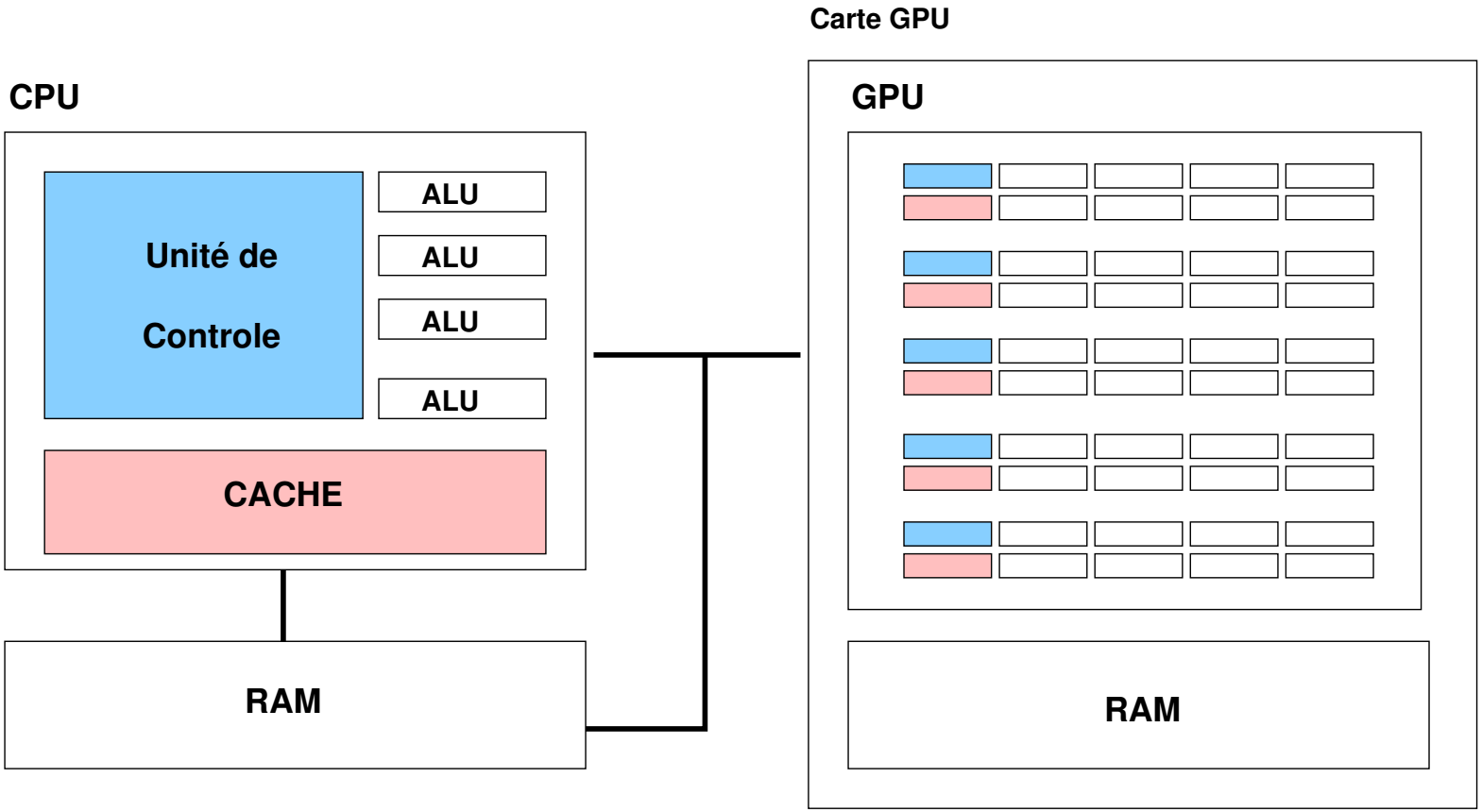
- permettant de d'augmenter les performances de calcul d'un système en exploitant des processeurs graphiques (GPU) ;
- repose sur les processeurs Tesla avec les architectures : Kepler, Fermi etc.
- support des cartes graphiques grand public (GeForce, Quadro) ;
- de 192 à 2800 processeurs sur une carte fille comportant une RAM propre importante (6Go par ex).

Le modèle de programmation CUDA

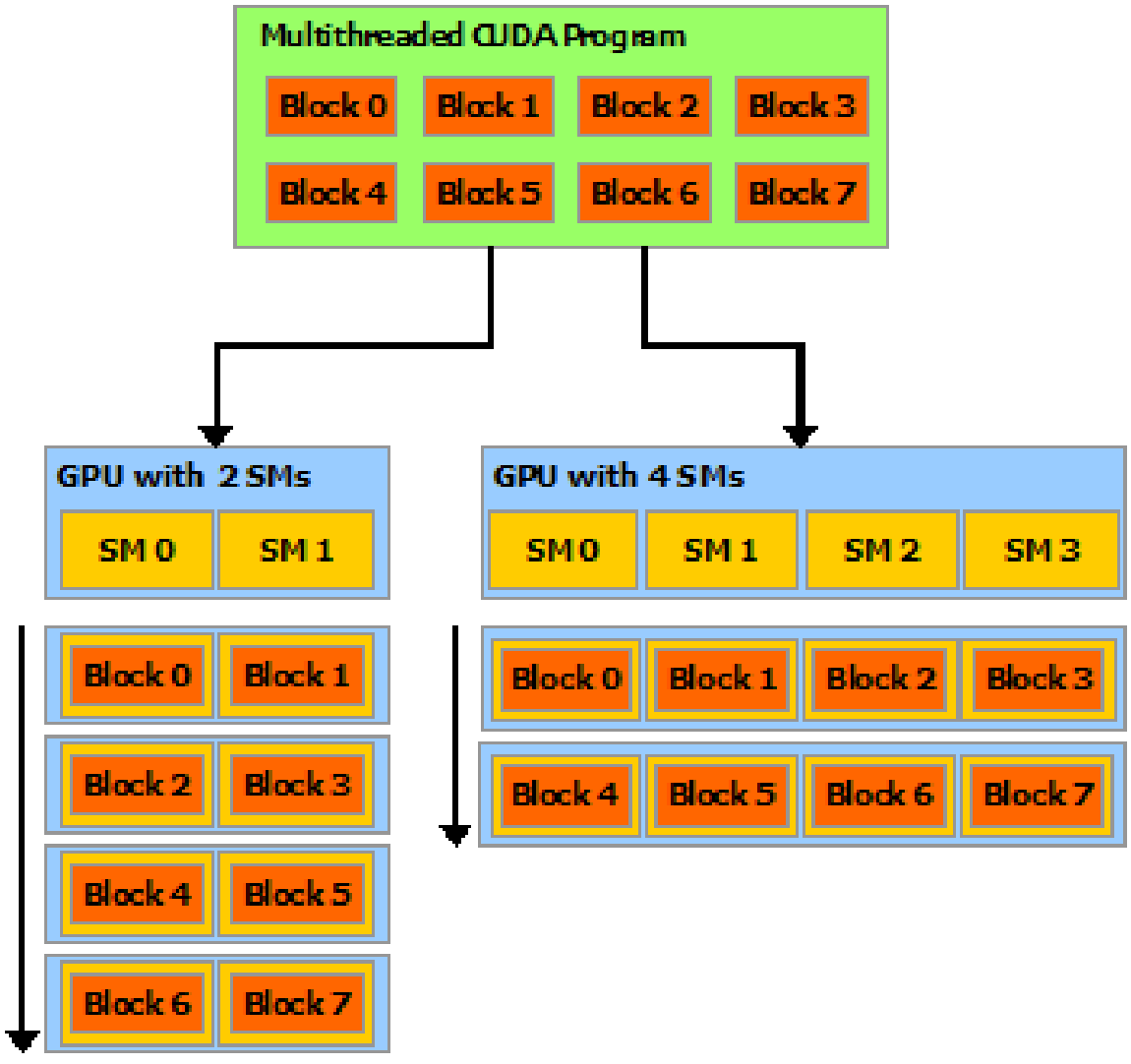




Le modèle de programmation CUDA



Le modèle de programmation CUDA



Exemple

La création de threads est peu coûteuse, on peut créer un thread pour additionner 2 valeurs et répéter l?opération pour un vecteur.

```
1  // Kernel definition
2  __global__ void VecAdd(float* A, float* B, float* C)
3  {
4      int i = threadIdx.x;
5      C[i] = A[i] + B[i];
6  }
7
8  int main()
9  {
10     ...
11     // Kernel invocation with N threads
12     VecAdd<<<1, N>>>(A, B, C);
13     ...
14 }
```

Rappel des notions abordées

- Notions de processus et de thread
- Partage de ressources entre processus (au sens large c-à-d y compris les thread)
- Problèmes de concurrence d'accès à une ressource
- Section critique, exclusion mutuelle
- Verrous, sémaphores, moniteurs
- Synchronisation des processus (`wait` et `notify` en Java)
- Problèmes de verrous mortels et de famine