Chap. 1

Expression des activités parallèles : processus et threads

Département d'Informatique
Université de Batna 2

Master s : ISI

Semestre : 1

Plan

- **Processus**
  - Généralités sur les processus
    - Définitions
    - Caractéristiques : Espace d'adressage, BCP
    - Commutation de contexte
  - Cas pratique : Processus sous Unix/Linux
- **Thread**
  - Généralités sur les threads
    - Principe
    - Pourquoi les threads
    - Implantation des threads : Utilisateur/ noyau
  - Cas pratique : Threads sous Unix/Linux & Threads en Java

GÉNÉRALITÉS SUR LES PROCESSUS

Processus

- **Définitions**
  - Programme :
    - Une entité statique
    - Une séquence d'instructions décrivant une tâche;.
  - Processus : est une abstraction d'un programme en exécution
    - Entité dynamique;
    - Représente une séquence d'actions qui matérialise un programme en exécution.
- **La conception et la réalisation d'un système d'exploitation (OS) repose sur ce concept de Processus**
- **Un OS doit**
  - Allouer les ressources aux processus.
  - Utiliser au maximum le CPU, tout en exécutant les processus à tour de rôle.
  - Assurer la communication et la synchronisation interprocessus.
  - Autoriser aux utilisateurs la création de processus

Processus

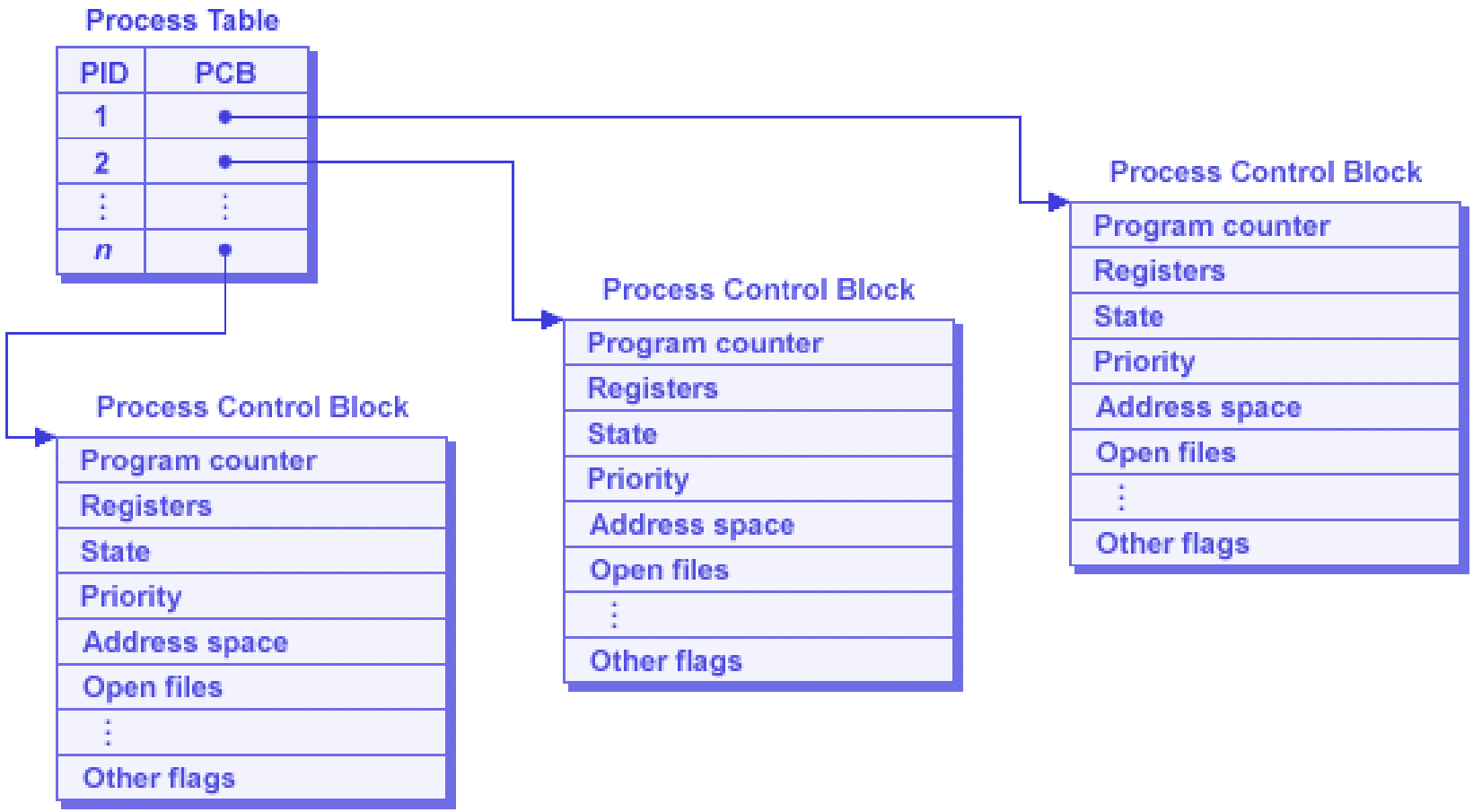
- **Que peut faire un processus ?**
  - Il peut créer d'autre processus
  - Il exécute des actions et communique avec d'autres processus Il peut posséder une ressource à titre exclusif ou bien la partager avec d'autre processus.
- **C'est le rôle de l'OS :**
  - d'assurer la gestion complète de la création, de la destruction, des transitions d'états d'un processus.
  - d'allouer un espace mémoire utile aux actions d'un processus.
  - D'assurer la synchronisation et la communication interprocessus.

Processus

- **Caractéristiques :**
  - **Bloc de contrôle de Processus (PCB)**
    - **PCB** (de l'anglais **Process Control Block**) est une structure de données qui décrit un processus dans OS
    - Un **PCB** regroupe toutes les informations nécessaires à la gestion d'un processus.
  - **Espace d'adressage**
    - En anglais (**address space**)
    - Il est divisé en trois espaces :
      - Espace de code
      - espace de données
      - et pile

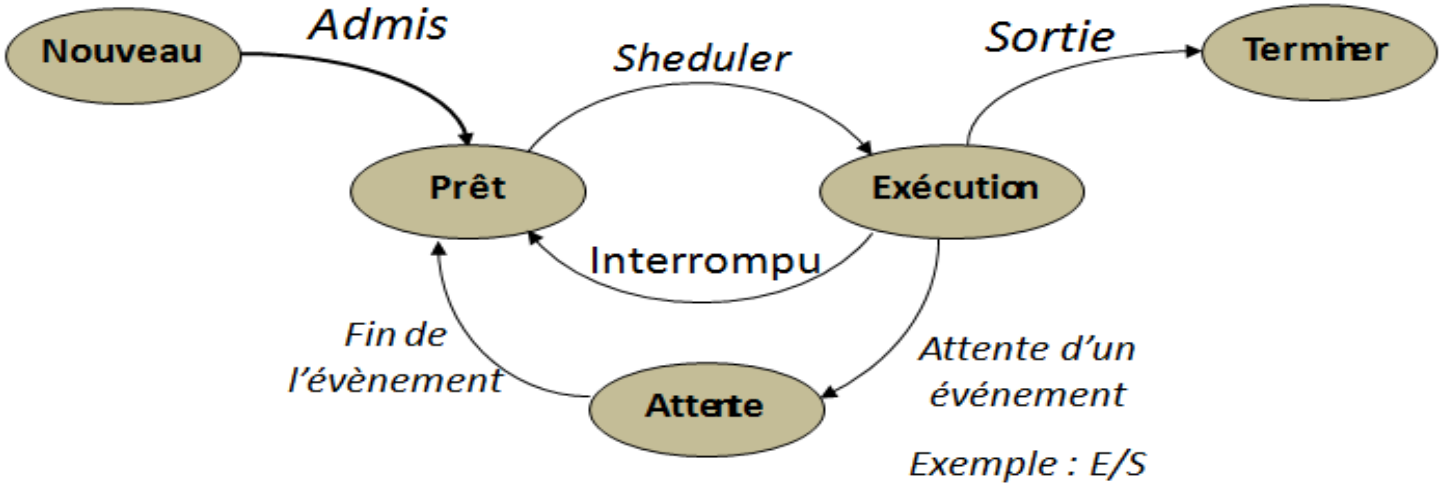
Processus

- **PCB**
  - Dans un OS, les processus sont généralement décrits par leurs PBCs accessibles à travers une table dite **Table des Processus**



Processus

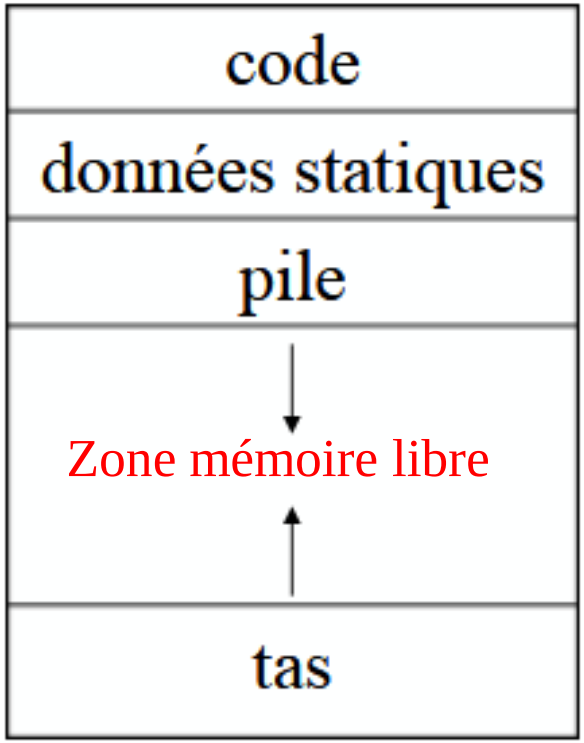
- **PCB**
  - Identificateur du processus (***PID***)
  - Contexte de l'unité centrale (***registers***)
    - Accumulateurs(Ax, Bx, ..etc), Registre d'instruction (RI) et compteur d'instruction (CØ), Registre d'état de CPU (registre PSW)
  - État du processus (***state***)
    - Nouveau, Prêt, Elu (en exécution), Bloqué (Endormi) et Terminé,



  - Priorité (***priority***)

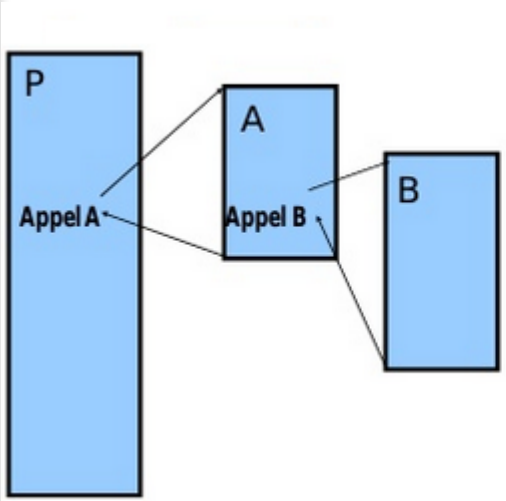
Processus

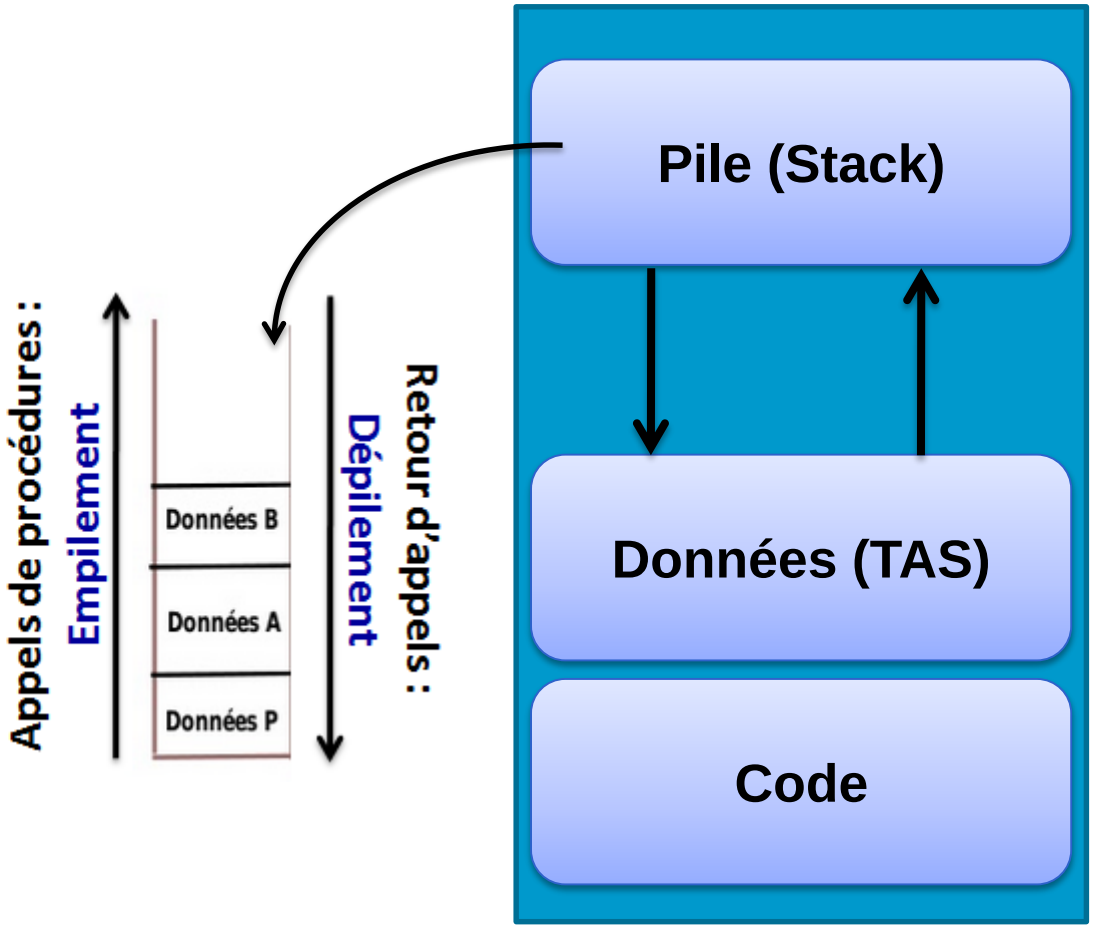
- **Espace d'adressage**
  - **code** : correspond aux instructions, en langage machine du programme à exécuter.
  - **Données statiques :** est un emplacement pour stocker les valeurs des variables manipulées par le processus
  - **Pile** : les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la pile.
  - **Tas** (heap-based buffer): est un emplacement de la mémoire utilisé pour les allocations dynamiques, par exemple les pointeurs.



Processus

- **Rôle de la pile**





Processus

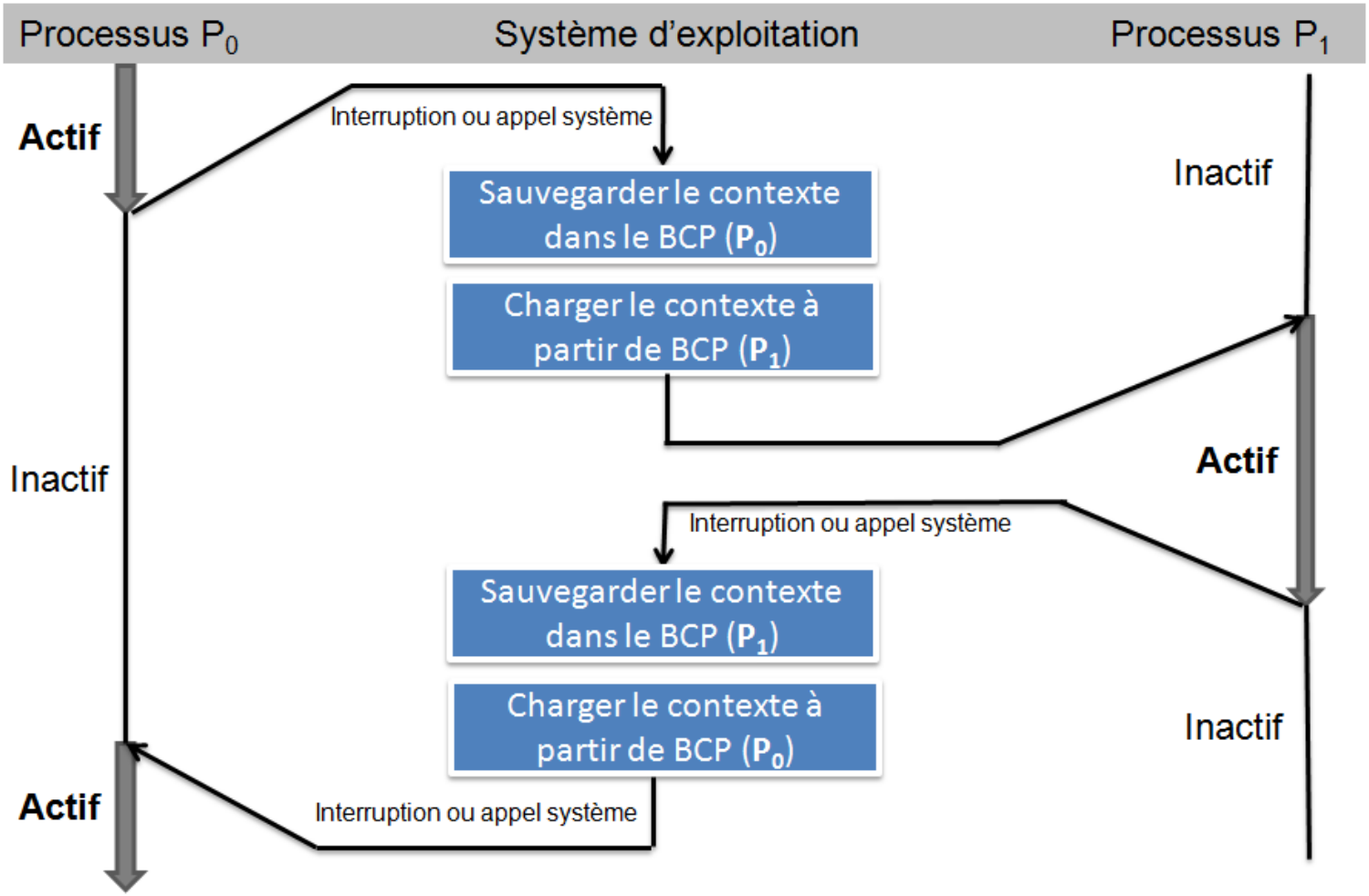
- **Droits d'accès d'un processus (*PSW : Program Status Word*)**
  - *PSW* Appelé aussi registre d'état.
  - décrivent les droits qu'a un processus pour accéder aux ressources à savoir :
    - ***Mode d'exécution*** (Master(système)/Slave(utilisateur) : traduit la possibilité d'exécuter des instructions privilégiées ou pas;
    - ***Indicateur d'utilisation de la mémoire virtuelle*** : traduit la possibilité de faire un adressage absolu (physique:@base+offset) ou pas;
    - ***Indicateur de masquage des interruptions*** : traduit le fait qu'un processus peut être interrompu ou non par certaines interruptions
    - ***Droits d'accès à la mémoire***

Processus

- **Planification des processus**
  - **Processus et Processeurs**
    - Un processeur est le composant électronique qui active un processus
    - Souvent le nombre de processus dépasse le nombre de processeurs
    - Conceptuellement, chaque processus possède son propre processeur virtuel, en réalité, le vrai processeur commute entre plusieurs processus.
    - C'est ainsi l'OS qui fournit un accès équilibré aux processeurs pour les processus (ordonnancement des processus).
    - Le partage d'un même processeur entre plusieurs processus requière l'exécution d'une opération dite commutation de contexte
  - **Commutation de contexte**
    - représente l'ensemble des actions utiles pour pouvoir reprendre l'exécution d'un processus interrompu.

Processus

- **Planification des processus**
  - **Commutation de contexte**



Cas Pratique

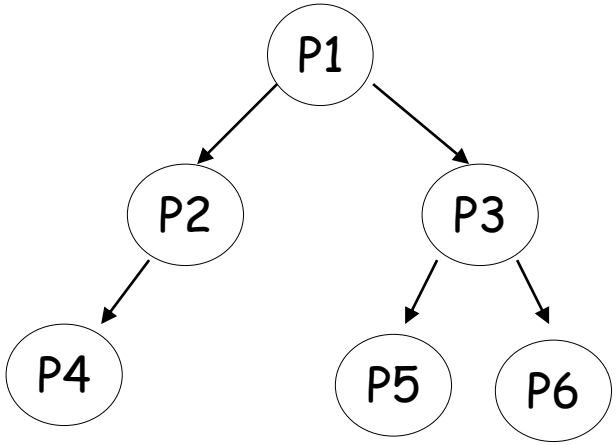
PROCESSUS SOUS UNIX/LINUX

Processus sous Unix/Linux

- **Hiérarchie des processus**
  - Le processus est l'entité d'exécution dans le système UNIX/Linux.
  - Au lancement du système, il n'existe qu'un seul processus, qui est l'ancêtre de tous les autres (processus 0: swapper/scheduler).
  - Chaque processus peut lancer lui-même d'autres processus.
  - Deux types de processus dans Unix/Linux
    - Processus Systèmes (ex :Swapper)
    - Processus Utilisateurs (exécution commande ou Application)

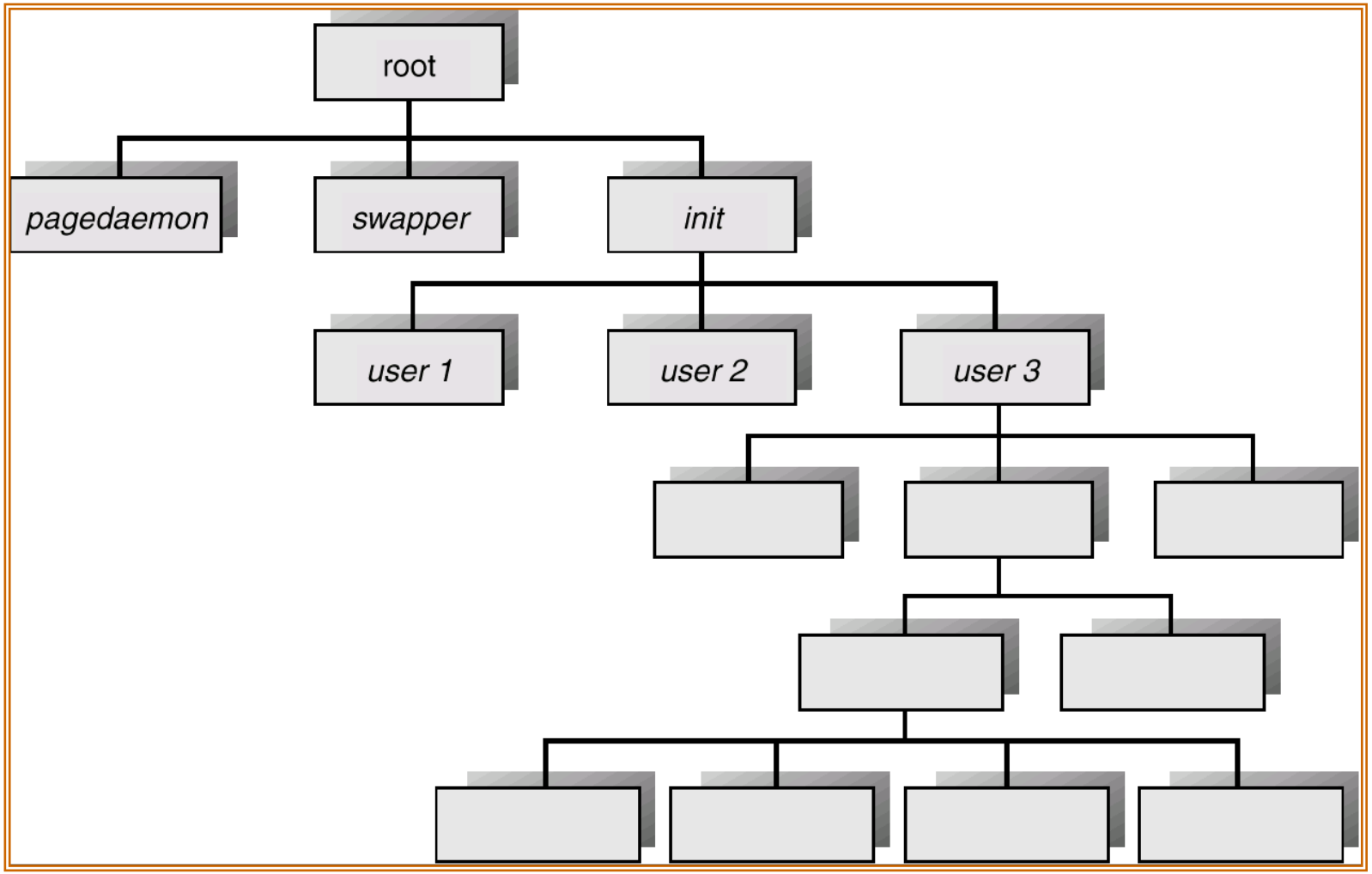
Processus sous Unix/Linux

- **Hiérarchie des processus**
  - Les processus des utilisateurs sont lancés par un interprète de commande (shell).
  - Ils peuvent eux même lancer ensuite d'autres processus
    - Le processus créateur est appelé le **père**
    - Les processus crées sont appelés les **fils**
  - Les processus sont structurés sous la forme d'une arborescence



Processus Processus sous Unix/Linux

- **Hiérarchie des processus**



Processus sous Unix/Linux

- **Propriétés et environnement d'un processus**
  - PID : Process ID
  - PPID : Parent Process ID
  - UID : User ID : Identifiant de l'utilisateur qui a lancé le processus
  - GID : Group ID : Identifiant du groupe auquel appartient l'utilisateur
  - STIME : correspond à l'heure de lancement du processus
  - TTY : correspond au nom du terminal à partir duquel s'est créé
  - TIME : correspond au temps déjà consommé
  - CMD : la commande qui a générée le processus
  - C : Facteur de priorité
  - Répertoire courant
  - Fichiers ouverts par le processus
  - ulimit la taille maximale des fichiers que le processus peut créer

Processus sous Unix/Linux

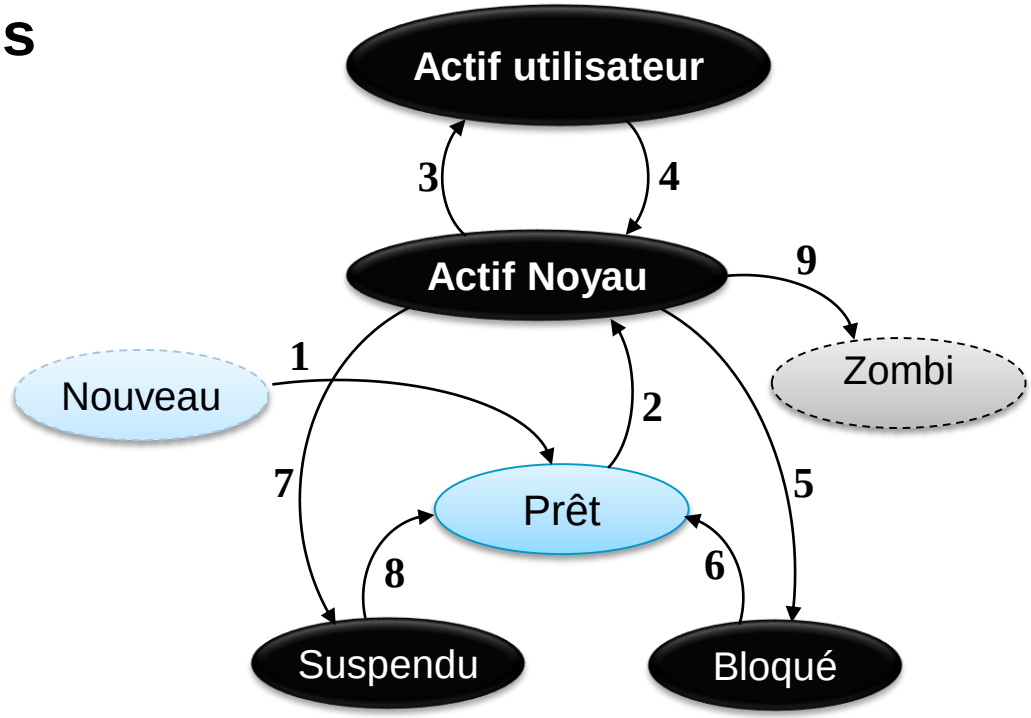
- **Propriétés et environnement d'un processus**

- La commande **ps** (***process status***) permet de visualiser les processus en cours d'exécution avec quelques informations d'environnement

```
$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
user        3139  3138  0 18:05 pts/0    00:00:00 -bash
user        4028     1  0 18:45 pts/0    00:00:00 dbus-l
user        4171  3139  0 18:54 pts/0    00:00:00 kate -
user        4748  3139  0 19:37 pts/0    00:00:00 kate -
user        4751  3139  0 19:37 pts/0    00:00:00 gedit
user        4845  3139  0 19:38 pts/0    00:00:00 ps -f
```

Processus sous Unix/Linux

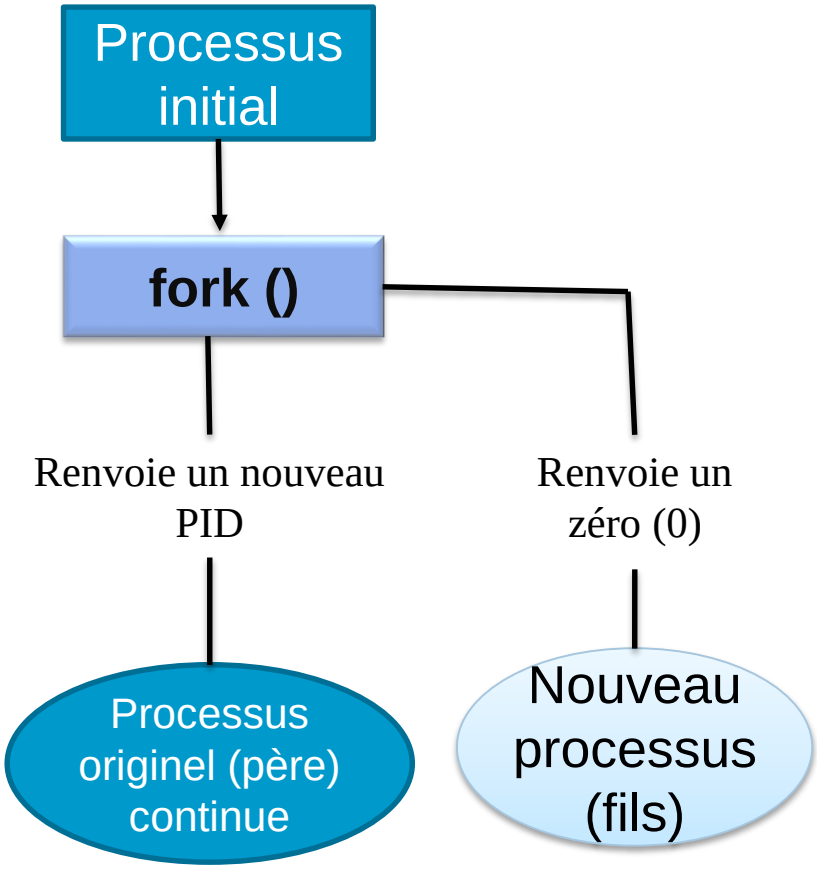
- **Etats d'un processus**
  - Nouveau
  - Actif
    - Actif noyau
    - Actif utilisateur
  - Prêt
  - En attente
    - Suspendu
    - Bloqué
  - Zombi



1. Allocation de ressources
2. Processus élu par l'ordonnanceur
3. Processus revient d'un appel système ou d'une interruption
4. Processus a fait un appel système ou a été interrompu
5. Processus se met en attente d'un événement
6. Événement attendu par le processus est arrivé
7. Processus suspendu par le signal SIGSTOP
8. Processus réveillé par le signal SIGCONT
9. Processus a fini sont exécution

Processus sous Unix/Linux

- **Création de processus**
  - Se réalise par duplication de processus par l'appel système fork()
    - #include <unistd.h>
    - pid t fork(void);
  - L'appel fork() retourne :
    - -1, si erreur,
    - 0, au fils,
    - pid du fils, retourné au père.
  - Suite à un fork(), le père et le fils poursuivent l'exécution du même programme.

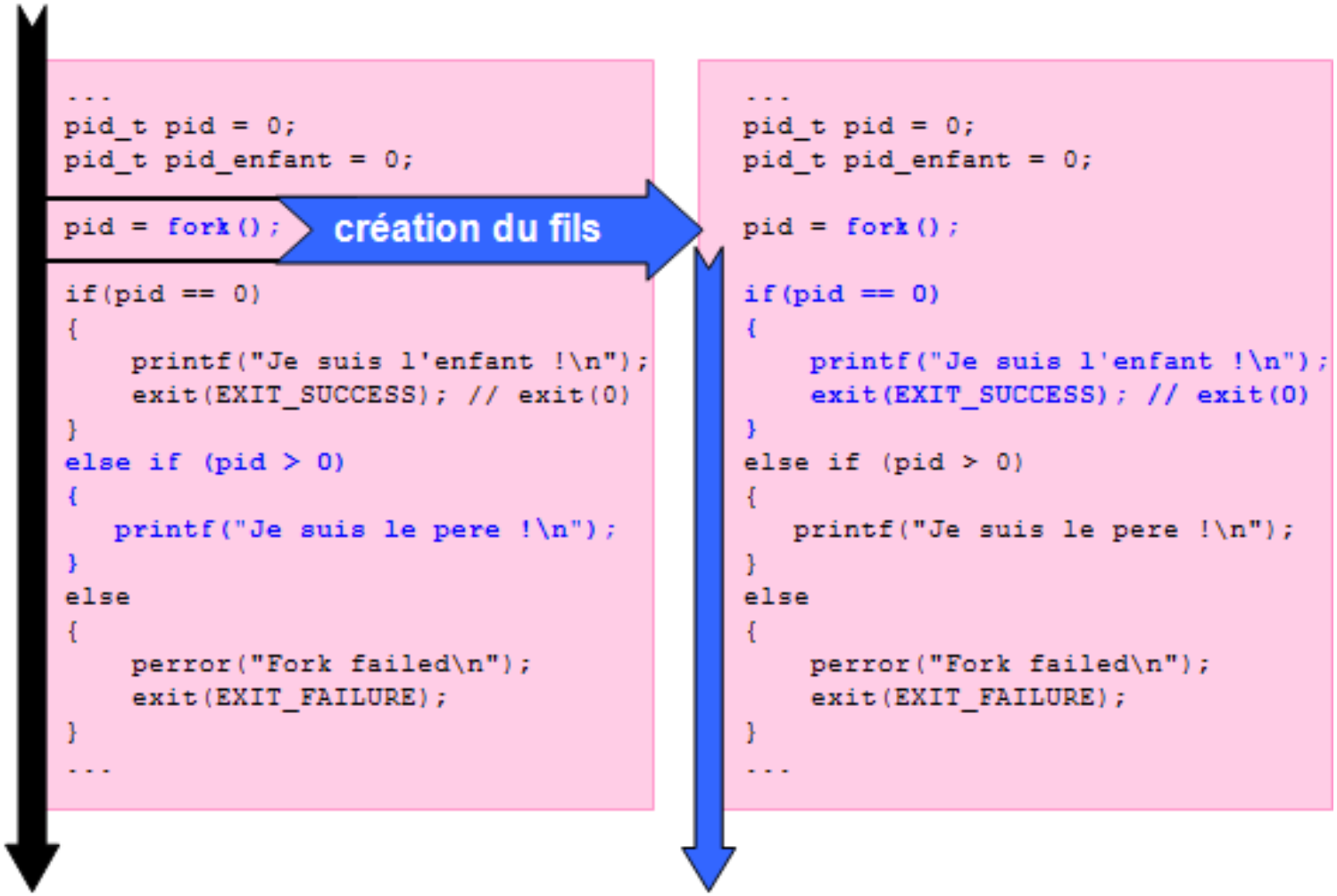


Processus sous Unix/Linux

- **Causes d'échec de l'appel système fork() :**
  - Le nombre maximum de processus en exécution par l'utilisateur est atteint (variable suivant les systèmes: 40 sous AIX, 50 sous ULTRIX, 997 sous Solaris 2.4).
  - Il ne reste pas suffisamment de mémoire système disponible pour dupliquer le processus.
  - Il n'y a pas assez d'espace de swap (swap space).

Processus sous Unix/Linux

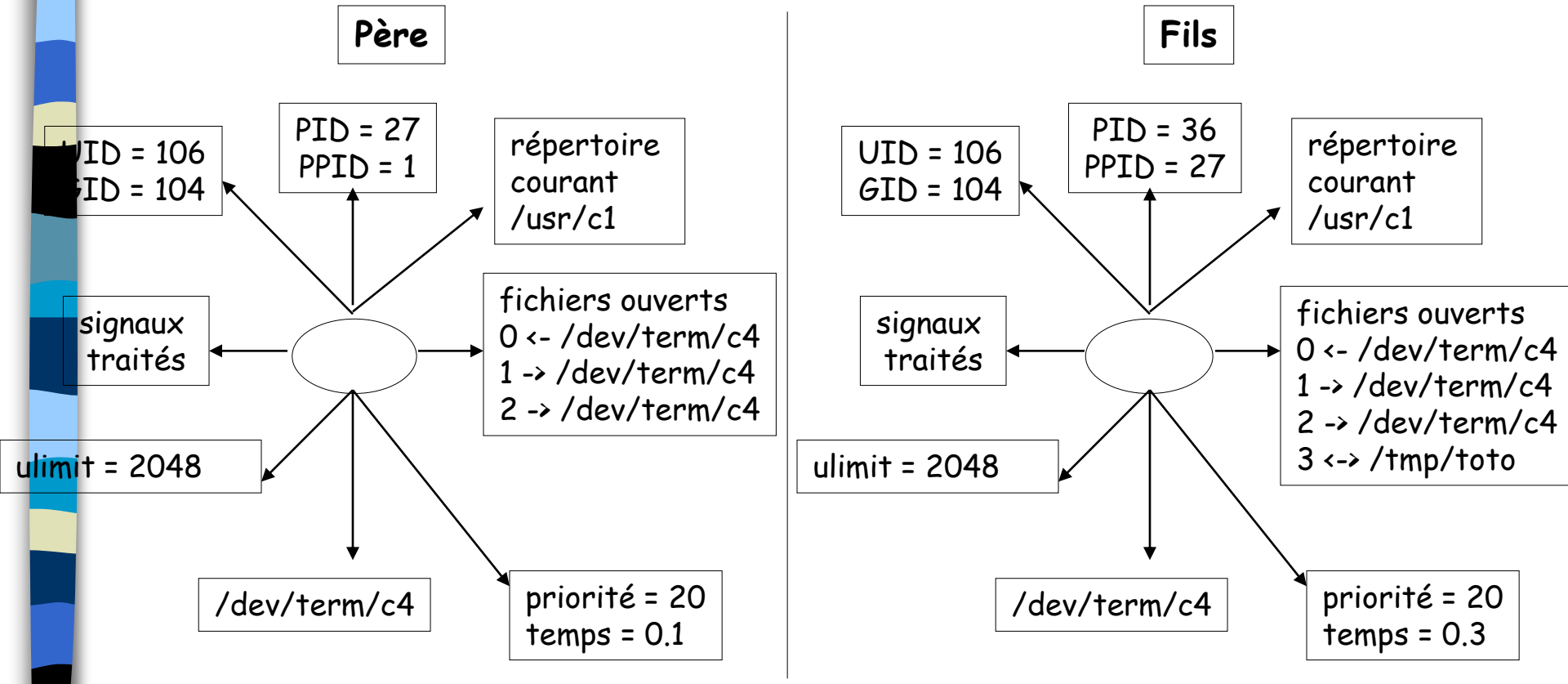
- **Création de processus**



Le même *programme* dans deux *processus* différents.

Processus sous Unix/Linux

- Le PPID est le PID du processus père
- Le processus fils hérite de tout l'environnement du processus père, sauf bien sûr du PID, du PPID et des temps d'exécution
- Le père du processus 36 est le processus 27, et celui de 27 est le processus 1
- Seul le fils 36 a ouvert le fichier /tmp/toto



Processus sous Unix/Linux

- **Synchronisation de processus**
  - wait()
    - Permet au processus d'attendre la terminaison d'un de ses fils
  - waitpid()
    - Permet au processus appelant d'attendre de manière sélective la terminaison d'un autre processus.
  - sleep() : Mise en sommeil d'un processus
    - **sleep(n)** suspend l'exécution du processus appelant pour une durée de n secondes.

Processus sous Unix/Linux

- **Exemple 1**

```
#include ‹stdio.h›
#include ‹unistd.h›
int main() {
        printf("Bonjour de : %d", getpid());
        int pid = fork();
        if (pid == 0) {
                printf("Je suis le FILS (pid = %d). %d\n", pid, getpid());  }
        else {
                printf("Je suis le PERE (pid = %d). %d\n", pid, getpid());  }
}
$ ./a.out
Bonjour de : 100250
Je suis le PERE pid = 100250
Je suis le FILS pid = 100251
```

- **Exemple 2**

```
int pid, a = 2, b=4;
 pid = fork();   /* fork another process */
 if (pid < 0)  exit(-1); /* fork failed */
 else if (pid == 0) /* child process */
        { a = 3; printf("%d\n", a+b); }
  else {
        wait(0); /* waiting child ending*/
        b = 1;
        printf("%d\n", a+b);
  }
```

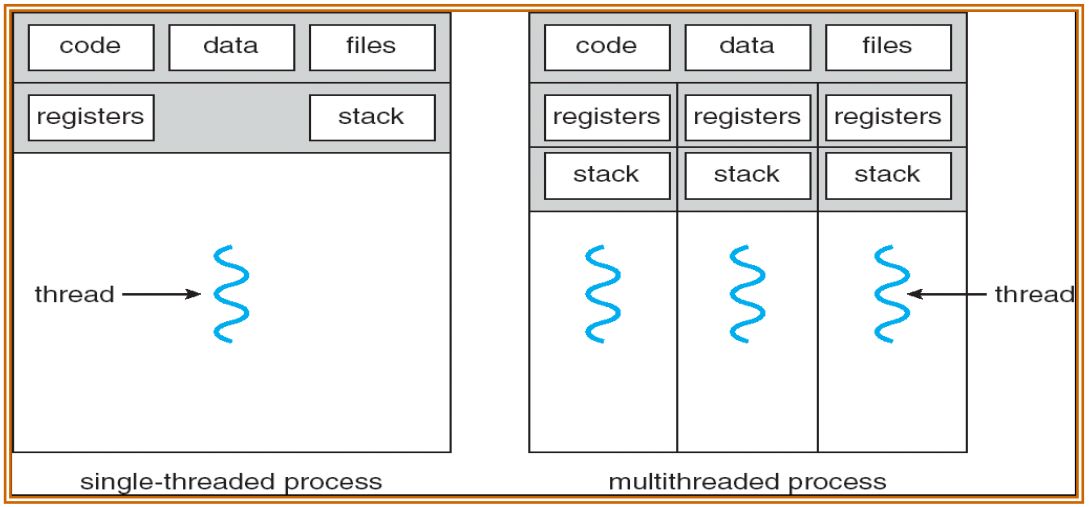
Que sera imprimé au terminal?

Inconvénients des processus

- Changement de contexte long (notamment pour les applications du type "temps réel" ou "multi média"),
- Pas de partage de mémoire (communications lentes)
- Pourquoi le changement de contexte est-il "long" :
  - 90% de ce temps est consacré à la gestion de la mémoire
- On introduit une nouvelle forme de processus : ceux-ci partagent la mémoire. Ce nouveau type d'activité s'appelle un thread

Threads

- **Principe**
  - Un thread est une subdivision du processus :
    - Un flot de contrôle dans un processus donc est une série d'instructions à exécuter.
  - Les threads permettent de dérouler plusieurs suites d'instructions, en *Parallèle*, à l'intérieur d'un même processus.
  - Le processus devient la structure d'allocation des ressources (fichiers, mémoire) pour les threads.



Threads

- **Pourquoi les threads?**
  - *Réactivité :* un processus peut être subdivisé en plusieurs threads. Exemple :
    - 1 thread dédié à l'interaction avec les usagers
    - 1 thread dédié au traitement des données
  - *Economie :* Contrairement aux processus, les threads sont légers :
    - ils partagent le même espace d'adressage,
    - La communication inter-thread occasionne peu de surcharge,
    - Le passage contextuel d'un thread à un autre est peu coûteux.
  - *Vitesse d'exécution* : Sur des machines parallèles, les threads peuvent s'exécuter en parallèle sur des CPU différentes

Threads

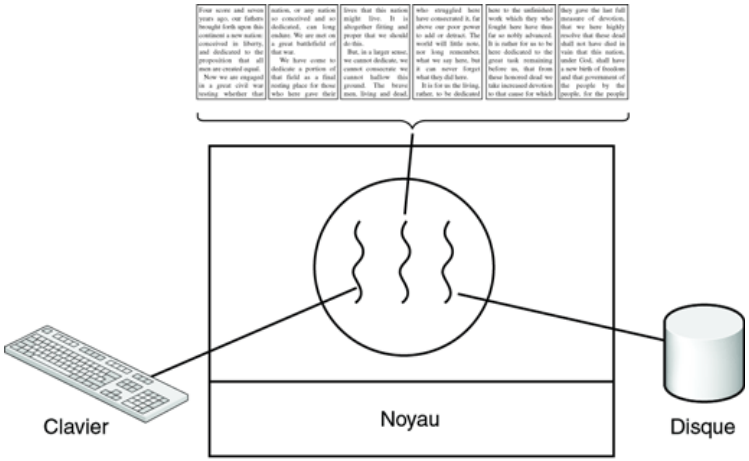
- **Pourquoi les threads?**
  - *Exemple*
    - Parallélisation simple des entrées-sorties
    - Chaque thread fait un appel bloquant, mais chaque appel est indépendant des autres :

Programmation Processus	Programmation threads

read(periph1, …);
read(periph2, …);
read(periph3, …);
 	
sauf option spéciale, chacun des appels bloque le suivant, empêchant le parallélisme des exécutions, imposant ainsi **un ordre**.
	pthread_create(read_periph1...);
	pthread_create(read_periph2...);
	pthread_create(read_periph3...);
	
	les trois requêtes se font en parallèle, **sans ordre**.

Threads

- **Utilisation des threads**
  - Fenêtres Multiples
  - Animation
  - Producteur/consommateur
  - Serveur avec plusieurs clients (web ou autres)
  - Recherche (base de données, web)

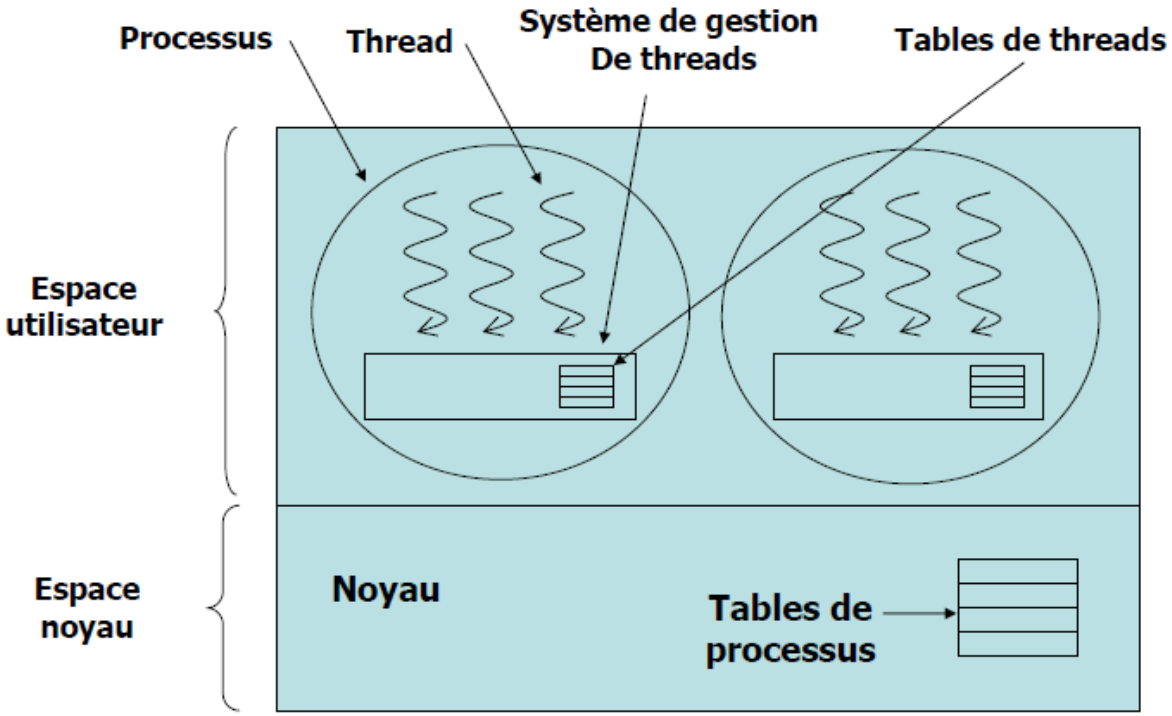


Threads

- **Implantation des threads**
  - au niveau du ***Processus*** qui l'accueille, il accède alors au processeur dans les quanta alloués à ce processus
    - **Threads utilisateur**
  - au niveau du ***Noyau***, il est alors ordonnancé indépendamment du processus dans lequel il a été créé,
    - **Threads noyau**

Threads

- **Threads utilisateurs**
  - On ne passe pas par l'OS pour gérer les threads.
  - Il y a des librairie qui le fait (ex: Java Virtual Machine)
  - Une table des thread par processus
  - Appel bloquant => on passe la main à un autre processus

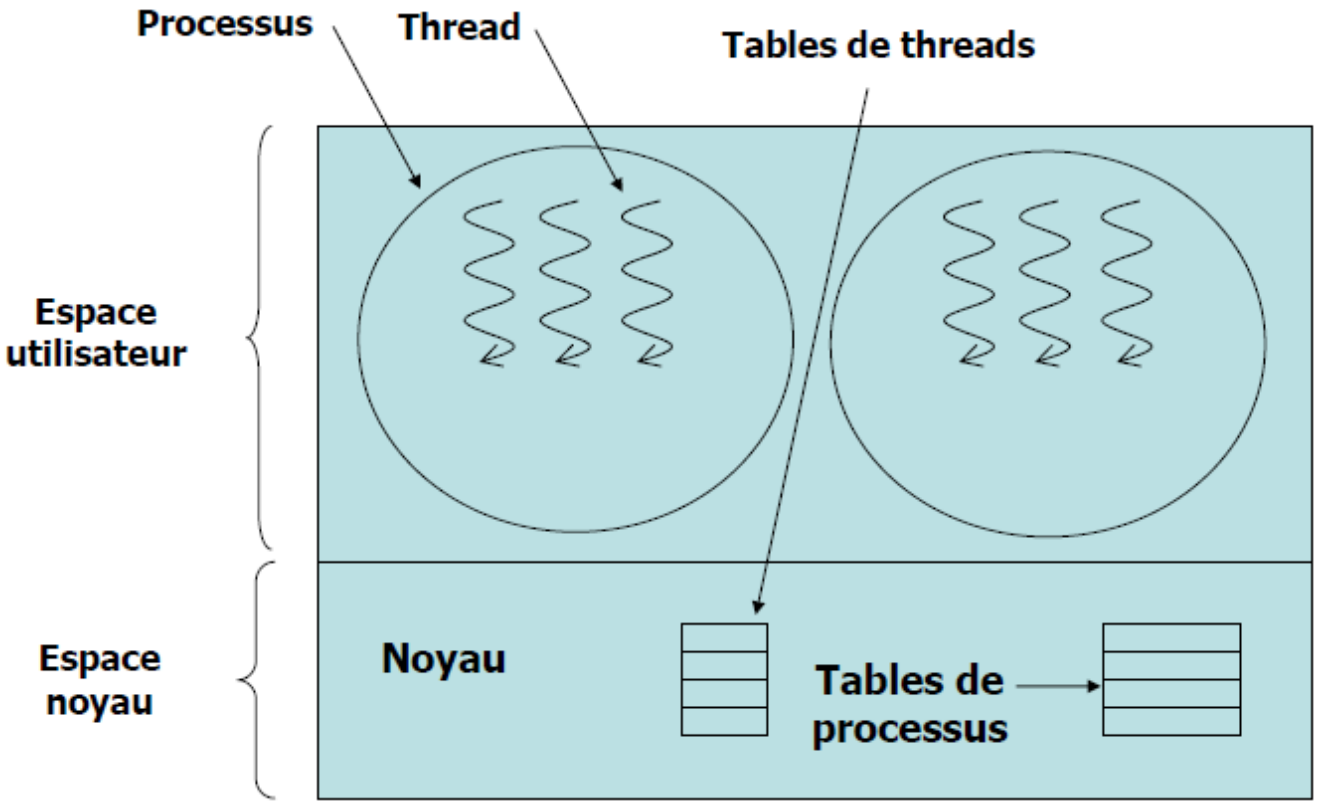


Threads

- **Threads utilisateurs**
  - **Avantages**
    - Transparence par rapport au système d'exploitation (portabilité)
    - Performance : La gestion des threads se fait à l'aide de procédure locales, pas d'appels système.
    - Algorithme d'ordonnancement des threads propre à chaque processus.
  - **Inconvénients**
    - Si un Thread se bloque(appel system bloquant. Ex : E/S), Tout le processus se bloque.
    - Si un thread est corrompu (Ex : Violation mémoire), Tout le processus est arrêté.
    - Les threads d'un même processus ne peuvent pas s'exécuter sur des processeurs différents.

Threads

- **Threads Noyau**
  - C'est l'OS qui gère les threads
  - On a une table des threads dans l'OS
  - les opérations sur les threads sont des appels au système



Threads

- **Threads Noyau**
  - **Avantages**
    - on peut affecter différents threads à différentes CPUs
    - Lorsqu'un thread est bloqué, on peut rendre la main à un thread du même processus
  - **Inconvénients**
    - Création/ gestion des threads nécessite des appel système ➔ lourdeur.

Threads

- **Dans le système Unix**
  - Les deux implémentation (Utilisateur et Noyau) sont supportées.
  - Espace utilisateur : Bibliothèque *pthreads*
    - Portable threads
    - Implémentation de la norme POSIX.1c
  - Espace noyau : appel system *clone()*
    - *Dans ce cas linux ne fait pas la différence entre processus et threads. Il les voix de la même manière.*

POSIX : sont des normes qui ont émergé d'un projet de standardisation des interfaces de programmation des logiciels destinés à fonctionner sur les variantes du système d'exploitation UNIX.

Threads

- **Comparaison entre threads utilisateur et noyau :**

Point de vue	Thread noyau	Thread utilisateur
Création d'un thread	Nécessite un appel système	Ne nécessite pas d'appel système, est donc moins que l'implémentation d'un thread noyau
Commutation entre deux threads	Assurée par le système avec changement du contexte du processus d'accueil	Commutation assurée dans la bibliothèque sans changement du contexte du processus d'accueil
Ordonnancement des threads	Chaque thread dispose des mêmes ressources CPU que les autres processus du système	Utilisation globale des ressources CPU limitée à celle du processus d'accueil.
Parallélisme	Sur une machine multiprocesseurs le système peut répartir les threads sur les différents processeurs	Les threads d'un même processus sont condamnés à s'exécuter sur le même processeur du processus d'accueil.

Threads en C sous Linux

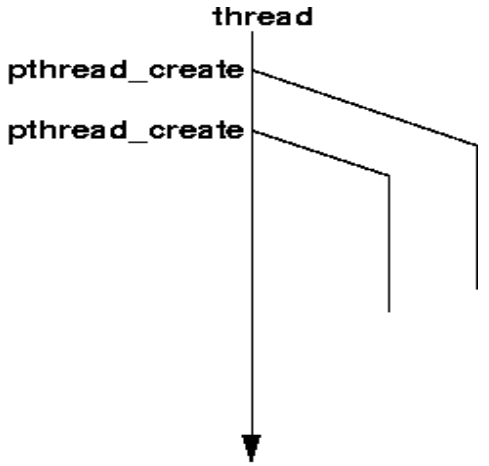
- ***Pthreads:***
  - Création et éliminations de threads
    - **`pthread_create`**
    - **`pthread_join`**
  - Déterminer l'id du thread courant
    - **`pthread_self`**
  - Terminaison d'un threads
    - **`pthread_cancel`**
    - **`pthread_exit`**
    - **`exit` [terminer tous les threads] , `ret` [terminer le thread courant]**
  - Synchroniser l'accès aux variables partagées
    - **`pthread_mutex_init`**
    - **`pthread_mutex_[un]lock`**
    - **`pthread_cond_init`**
    - **`pthread_cond_[timed]wait`**

Threads en C sous Linux

- ***Pthreads:***
  - Création d'un thread
    - `int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *nomfonction, void *arg);`
      - pthread_create() crée un thread qui exécute la fonction **`nomfonction`** avec l'argument **`arg`** et les attributs **`attr`**.
      - Les attributs permettent de spécifier la taille de la pile ,la priorité, la politique de planification, etc.
  - Eliminations d'un thread
    - `int pthread_join (pthread_t *thread, void* *valeur_de_retour);`
      - Suspend l'appelant jusqu'à ce que le thread de l'identifiant *thread termine
      - **`*valeur_de_retour`**

Threads en C sous Linux

- ***Pthreads:***

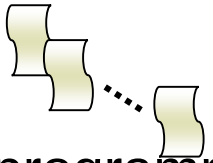


– *Exemple de Création de threads*

```
#include <pthread.h>
#include <stdio.h>
void *nomfonction (void * num) {
      int i;
      For (i=0; i<10;i++){
            printf ("Thread%d : Hello World ! Mon identifiant: %d \n", num,
            pthread_self());
      }
}
void main (){
  pthread_t  thread0, thread1;
  pthread_create (&thread0, NULL, nomfonction, (void *) 0);
  pthread_create (&thread1, NULL, nomfonction, (void *) 1);
  pthread_join (thread0, NULL);
  pthread_join (thread1, NULLl);
}
```

Threads en Java

- **Caractéristiques des Threads Java**
  - Les threads sont des processus indépendants. Aussi appelés Tâches.



  - Les threads permettent d'exécuter plusieurs programmes indépendants les uns des autres. Ceci permet une exécution parallèle de différentes tâches de façon autonome.
  - La gestion des multithreads est intégrée dans le langage Java (dans la Machine Virtuelle)
  - Java met à la disposition des programmeurs les moyens permettant de synchroniser les threads

Threads en Java

- **Création de threads en Java se fait par 2 méthodes**
  - *Hérité de la classe Thread*
  - *Par la méthode Runnable (interface runnable).*
- ***Créer et démarrer un thread (extends Thread)***
  - Hériter de la classe Thread et surcharger la méthode **run**.
  - La fonction **main** va créer deux threads et les démarrer.

```
class essai {
 public static void main (String args[]) {
     Thread T1 = new MonThread(" T1 ");
     Thread T2 = new MonThread(" T2 ");
     System.out.println("Lancement de T2");
    T2.start(); // start appelle la methode run de T2
    System.out.println("Lancement de T1");
    T1.start(); // start appelle la methode run de T1
    while (true) ;
  }}
```

Threads en Java

- ***Créer et démarrer un thread (extends Thread)***
  - La classe MonThread hérite de Thread

```
class MonThread extends Thread {
    public MonThread(String str) {
        super(str);  //
    }
    public void run() {
        System.out.println("Exécution de " + getName());
    }
}
```

Exécution 1	Exécution 2
Lancement de T2 Exécution de T2 Lancement de T1 Exécution de T1	Lancement de T2 Lancement de T1 Exécution de T2 Exécution de T1

Threads en Java

- ***Créer et démarrer un thread (Runnable)***
  - La méthode run() de l'objet Thread appelle celle de l'objet Runnable
  - Ceci permet à des threads de s'exécuter dans n'importe quel objet, sans utiliser l'héritage. On invoque leur méthode run ():

```
class essai2 {
    public static void main (String args[]) {
        Action A1 = new Action(" A1 ");
        Action A2 = new Action(" A2 ");
        System.out.println("Lancement de A2");
        A2.run();
        System.out.println("Lancement de A1");
        A1.run();
        while (true) ;
    }
}
```

Threads en Java

- ***Créer et démarrer un thread (Runnable)***
  - On utilise des objets qui ne sont pas des threads. On invoque leur méthode run ():

```
class Action  {
        String local;
        public Action (String str) {
             local = str;
        }
        public void run() {
           System.out.println("Exécution de " + local );
        }
}
```

Exécution
Lancement de A2 Exécution de A2 Lancement de A1 Exécution de A1

Threads en Java

- ***Créer et démarrer un thread (Runnable)***
  - On transforme ces objets en threads. On appelle start qui appellera run() :

```
class essai2 {
    public static void main (String args[]) {
        Action A1 = new Action(" A1 ");
        Action A2 = new Action(" A2 ");
        System.out.println("Lancement de A2");
        new Thread(A2).start();
        System.out.println("Lancement de A1");
        new Thread(A1).start();
        while (true) ;
    }
}
```

Threads en Java

- ***Créer et démarrer un thread (Runnable)***

```
class Action implements Runnable
{
    String local;
    public Action (String str) {
       local = str;
    }
    public void run() {
       System.out.println("Exécution de " + local );
    }
}
```

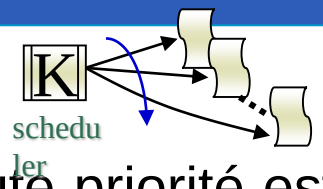
Exécution
Lancement de A2 Lancement de A1 Execution de A2 Execution de A1

Threads en Java

- **Comparaison**

	Avantages	Inconvénients
extends java.lang.Thread	Chaque thread a ses données qui lui sont propres.	On ne peut hériter d'une autre classe.
Implements java.lang.Runnable	L'héritage reste possible. On peut implémenter autant d'interfaces souhaitées.	Les données de la classe sont partagés par tous les threads qui y sont basés. Parfois, cela peut-être souhaité.

Threads en Java



- **Ordonnancement de threads Java**
  - En général, le thread exécutable avec la plus haute priorité est activé (running).
    - Chaque thread possède une priorité comprise entre 1 (plus petite priorité) et 10 (plus grande priorité):

    Thread.MIN_PRIORITY ≤ thread JAVA ≤ Thread.MAX_PRIORITY
    - Par défaut, un thread reçoit la priorité Thread.NORM_PRIORITY (de valeur = 5).
    - Chaque thread hérite de la priorité du thread qui l'a créé.
  - Java est priority-preemptive
    - Si un thread de haute-priorité se réveille, et un thread de basse priorité s'exécute, Alors le thread de haute priorité s'exécute immédiatement.
  - Les threads JAVA sont ordonnancés en temps partagés.
  - Lors de la compilation de source JAVA, un ordonnanceur simpliste est implémenté dans l'application.

Threads en Java

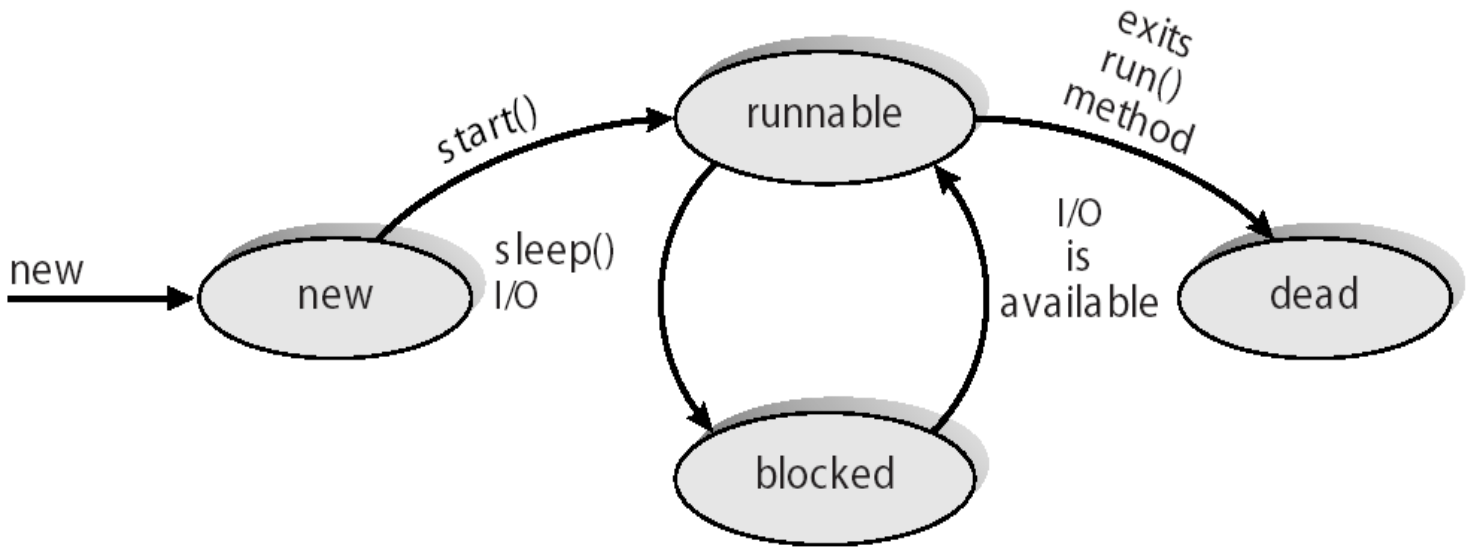
- **Stratégie d'attribution des priorités**
  - Threads qui ont beaucoup de travail à réaliser, devrait avoir une priorité plus faible.
  - Donner une priorité élever pour les tâches courtes.
  - Donner aux threads devant faire des I/O une haute priorité.
    - se réveiller, calculer immédiatement les données, attendre de nouveau les I/O.
- **Condition de la course des threads**
  - Deux threads modifient simultanément un objet
  - Les deux threads "courent" pour stocker leurs valeurs
  - Au final, le dernier "gagne la course"
  - (En fait, les deux perdent)
- **La solution est alors la synchronisation de threads**
  - On discutera après

Threads en Java

- **Famine d'un thread**
  - Si un thread de haute priorité n'est jamais bloqué, Alors tous les autres threads seront en attente de CPU, et ne travailleront jamais.
  - Prudence au niveau d'attribution de priorités aux threads.

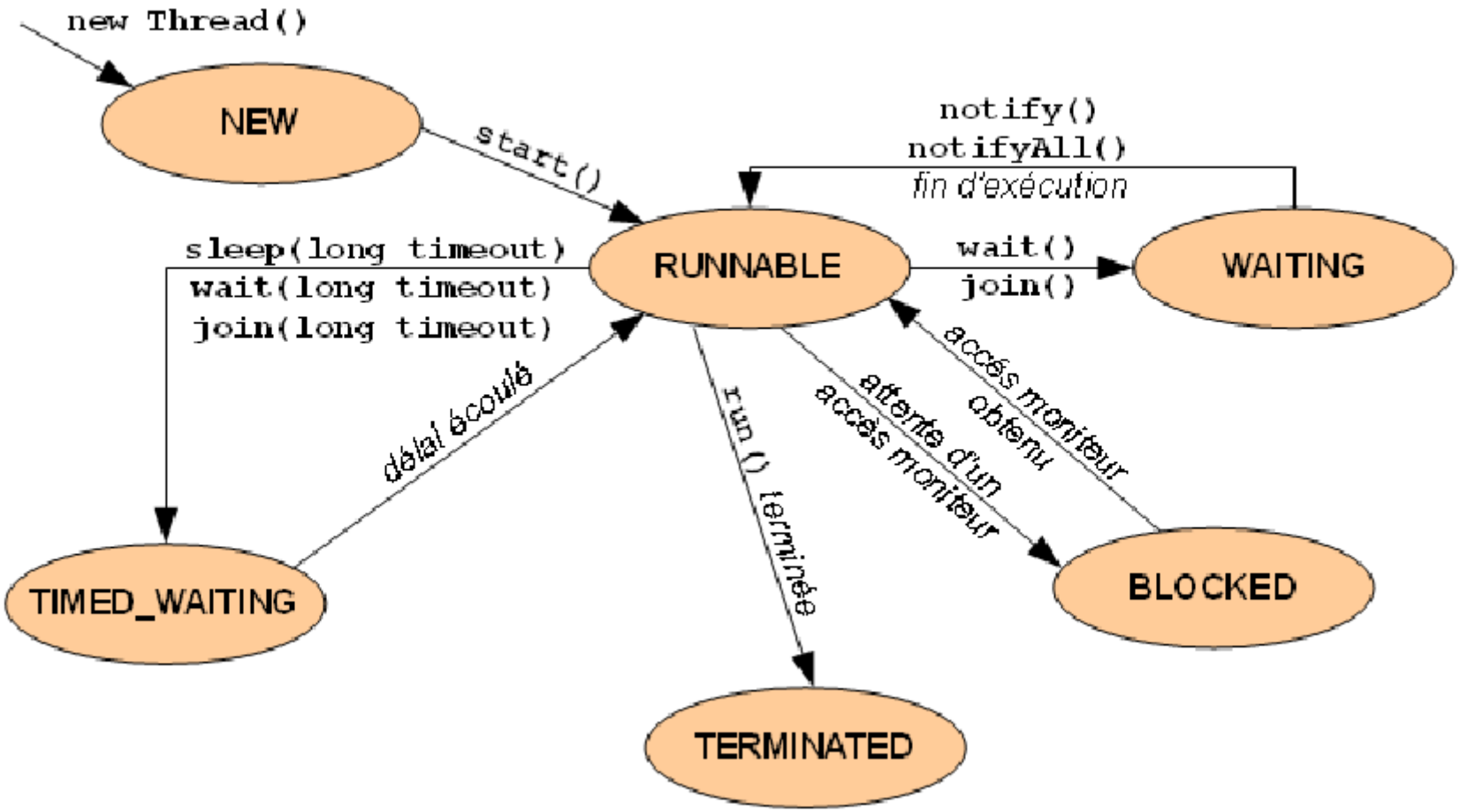
Threads en Java

- **Vie et mort d'un Thread en Java**



Threads en Java

- **Vie et mort d'un Thread en Java**



Threads en Java

- **Vie et mort d'un Thread en Java**
  - **Activation** :
    - la méthode start() appelle la méthode run() (start() n'est pas bloquante dans le thread appelant).
  - **Endormi ou bloqué :**
    - après sleep() : endormi pendant un intervalle de temps (ms)
    - suspend() endort le Thread mais resume() le réactive
    - une entrée/sortie bloquante (ouverture de fichier, entrée clavier) endort et réveille un Thread
  - **Mort** :
    - si stop() est appelé explicitement
    - quand run() a terminé son exécution

Threads en Java

- **Exercice :**

  Ecrire une classe Ping qui réalise un Thread qui affiche « Ping » à intervalle irrégulier Ecrire une classe Pong qui réalise un Thread qui affiche « Pong » à intervalle irrégulier Ecrire une classe Go qui lance les Threads Ping et Pong

Threads en Java

- **Solution de l'Exercice :**

```
import java.io.*;
class Ping extends Thread{
    public void run(){
        try{
            while (true){
                System.out.println(« Ping »);
                sleep((long)500*Math.random());
                }
            }
        catch(InterruptedException e){ return;}
        }
    }
```

Threads en Java

- **Solution de l'Exercice :**

```
import java.io.*;
class Pong extends Thread{
    public void run(){
        try{
            while (true){
                System.out.println(« Pong »);
                sleep((long)500*Math.random());
                }
            }
        catch(InterruptedException e){ return;}
        }
    }
```

Threads en Java

- **Solution de l'Exercice :**

```
import java.io.*;
class Go {
    public static void main(String args[])
    {
        Ping p1 = new Ping();
        Pong p2 = new Pong();
        p1.start();
        p2.start();
    }
```

Threads en Java

- **TP : *Démarrage de threads simples (avec la méthode « hérite de la classe thread »)***

Ecrire un programme JAVA qui démarre deux threads:

Le premier qui attend 1 seconde avant de remplir un entier en mémoire partagée avec la valeur 42, Le deuxième qui examine en boucle la valeur de cet entier et attend que cette valeur devienne égale à 42.

Pour ce faire, il faut créer deux classes (une par processus) qui chacune sont des sous-classes de Thread, et une autre classe qui contient un main lançant les deux processus. L'entier en mémoire partagée sera en fait une "classe enveloppante" pour les entiers, on pourra utiliser par exemple la classe suivante:

Threads en Java

```
public class UnEntier {
int val;
public UnEntier ( int x ) {
val = x;
}
public int LireVal () {
return val;
}
public void EcrireVal ( int x ) {
val = x;
}
}
```

Threads en Java

```
public class Exo1 {
public static void main(String[] args) {
        UnEntier   i = new UnEntier(0);
        new Proc1(i).start() ;
        new Proc2(i).start() ;
      }
}
```

Threads en Java

```
public class Proc1 extends Thread {
 UnEntier   refI;
 Proc1(UnEntier ent) {
    refI = ent;
 }
 public void run() {
       try {
          System.out.print("\nP1 starts...");
          sleep(100);
          refI.setValue(42);
          System.out.print("\nP1 ends...");
       } catch (InterruptedException e) { return ; }
  }
}
```

Threads en Java

```
public class Proc2 extends Thread {
    UnEntier  refI;
    Proc2 (UnEntier ent) {
       refI = ent;
    }
    public void run() {
     try {
     System.out.print("\nP2 starts...");
     while (refI.LireVal () != 42) {
     System.out.print("\nP2 waits...") ;
     sleep (5) ;
            }
            System.out.print("\nP2 ends...");
     } catch (InterruptedException e) {
     return ;
     }
    }
}
```