Module 3 - Fils (Threads)

Lecture: Chapitre 4

Objectif:

- **Comprendre le concept de fils et sa relation avec le processus**
- **Comprendre comment le systèmes d'exploitations gèrent et utilisent les fils.**

Sujets

- **Le fil d'exécution chez les processus**
- **Multi-fils versus fil unique (le thread)**
  - Les fils niveau usager et les fils niveau noyau
- **Les défis du « Threading »**
- **Exemples de fils**

Caractéristiques des processus

- **Unité de possession de ressources - un processus possède:**
  - Un espace virtuel adressable contenant l'image du processus
  - D'autres ressources (fichiers, unités E/S...)
- **Unité d'exécution (dispatching) - un processus s'exécute le long d'un chemin parmi plusieurs programmes**
  - L'exécution s'imbrique parmi l'exécution d'autres processus
  - Le processus possède un état d'exécution et de priorité utilisé pour l'ordonnancement

Caractéristiques des Processus

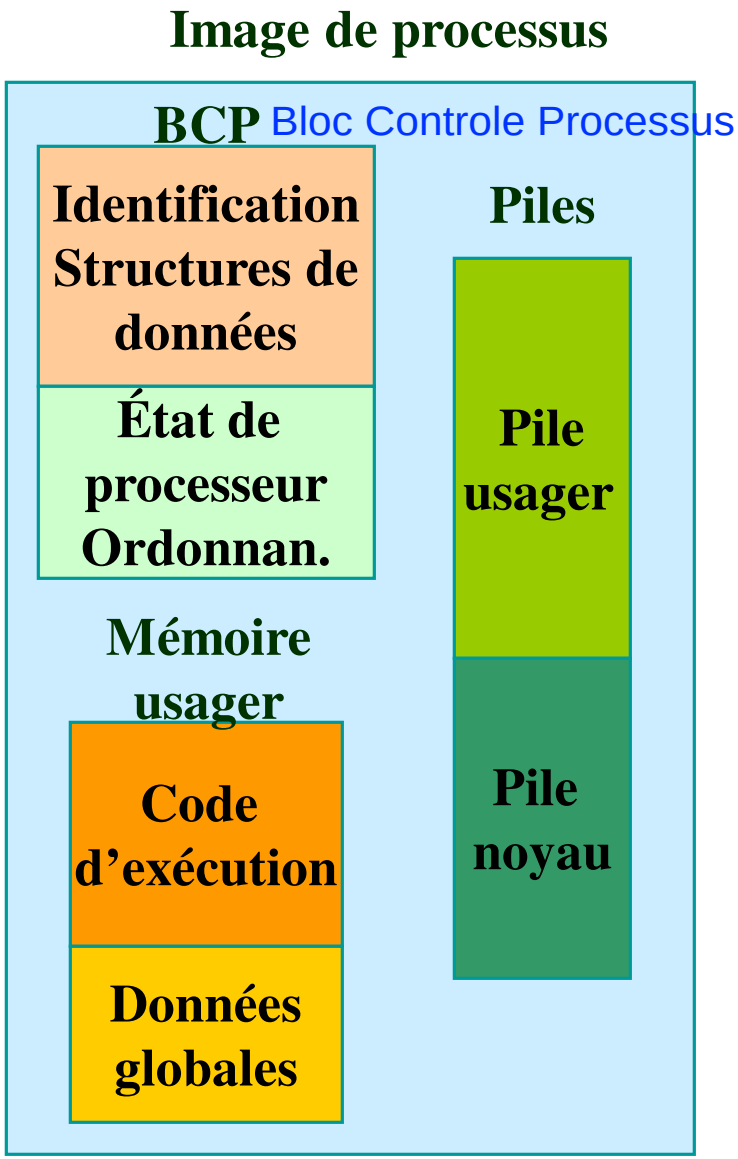
- **Possède sa mémoire, ses fichiers, ses ressources, etc.**
- **Accès protégé à la mémoire, fichiers, ressources d'autres processus**

Caractéristiques des processus

- **Ces 2 caractéristiques sont perçues comme étant indépendantes de certains SE**
- **L'unité d'exécution est habituellement désigné comme un fil d'exécution**
- **L'unité de possession de ressources est habituellement désigné comme un processus**

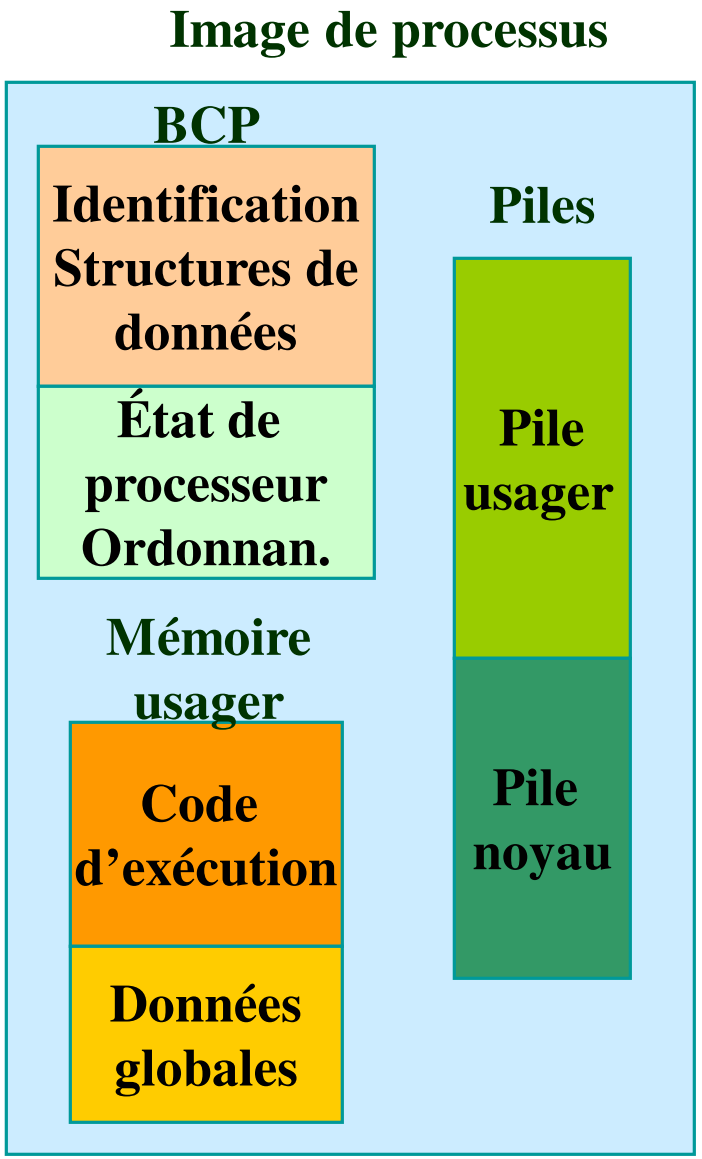
Unité de possession de ressources

- **Associée aux composantes suivantes de l'image d'un processus**
  - La partie du BCP qui contient l'information d'identification et les structures des données
  - La mémoire contenant le code d'exécution
  - La mémoire contenant les données globales



Unité d'exécution

- **Associée aux composantes suivantes de l'image d'un processus**
  - BCP
    - **L'état du processeur**
    - **Les structures d'ordonnancement**
  - Piles



Sujets

- **La fil d'exécution chez les processus**
- **Multi-fils versus fil unique (le thread)**
  - Les fils niveau usager et les fils niveau noyau
- **Les défis du « Threading »**
- **Exemples de fils**

Plusieurs fils(thread) sont considerer comme un flot

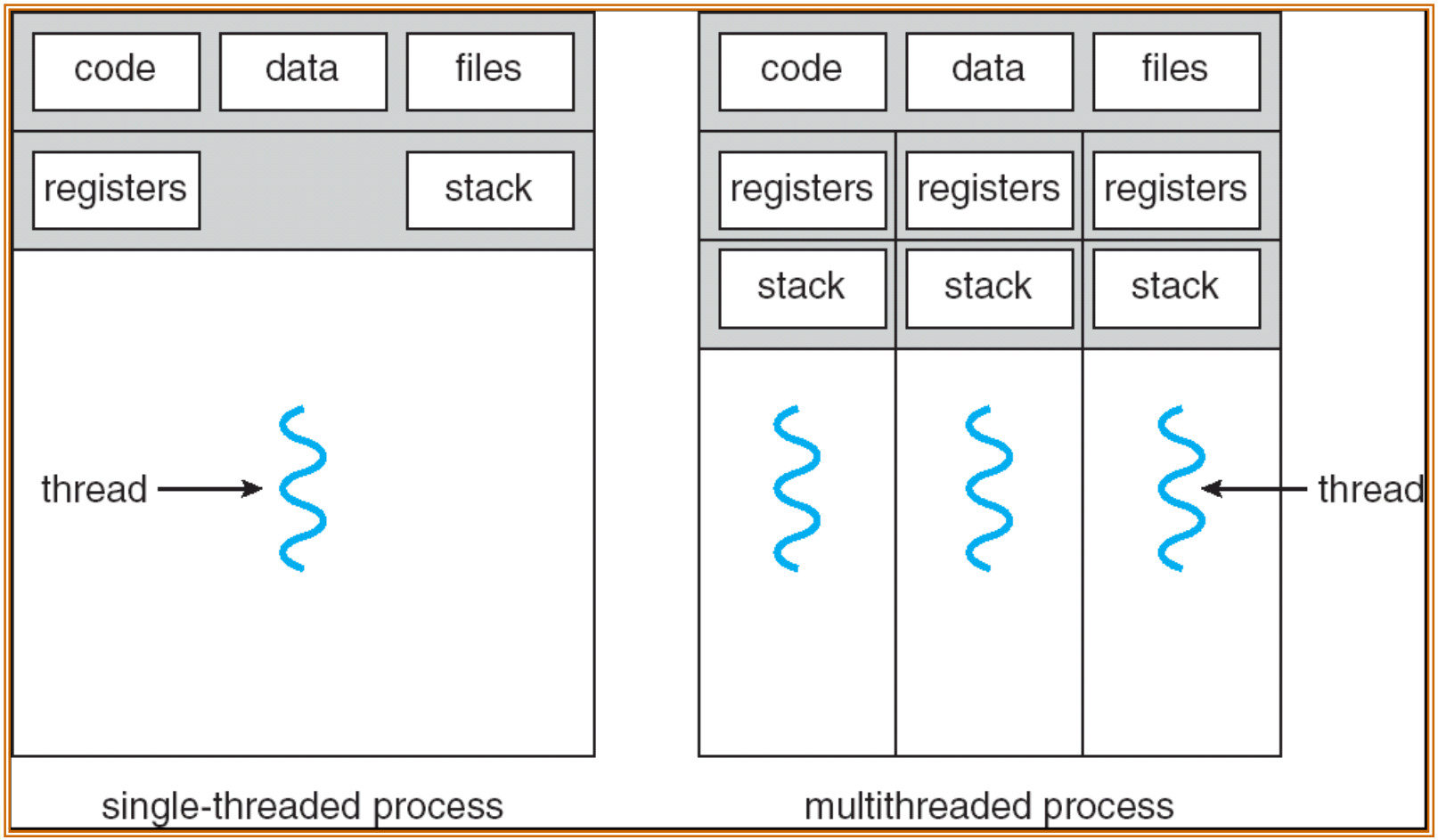
Fils = Flots = threads = lightweight processes

- **Un thread est une subdivision d`un processus**
  - Un fil de contrôle dans un processus
- **Les différents threads d'un processus partagent l'espace adressable et les ressources d'un processus**
  - lorsqu'un thread modifie une variable (non locale), tous les autres threads voient cette modification
  - un fichier ouvert par un thread est accessible aux autres threads (du même processus)

Exemple

- **Le processus MS-Word implique plusieurs threads:**
  - Une interaction avec le clavier
  - Un affichage de caractères sur la page impliquée
  - La sauvegarde régulière du travail fait
  - Le contrôle d'orthographe
  - Etc.
- **Ces threads partagent tous le même document**

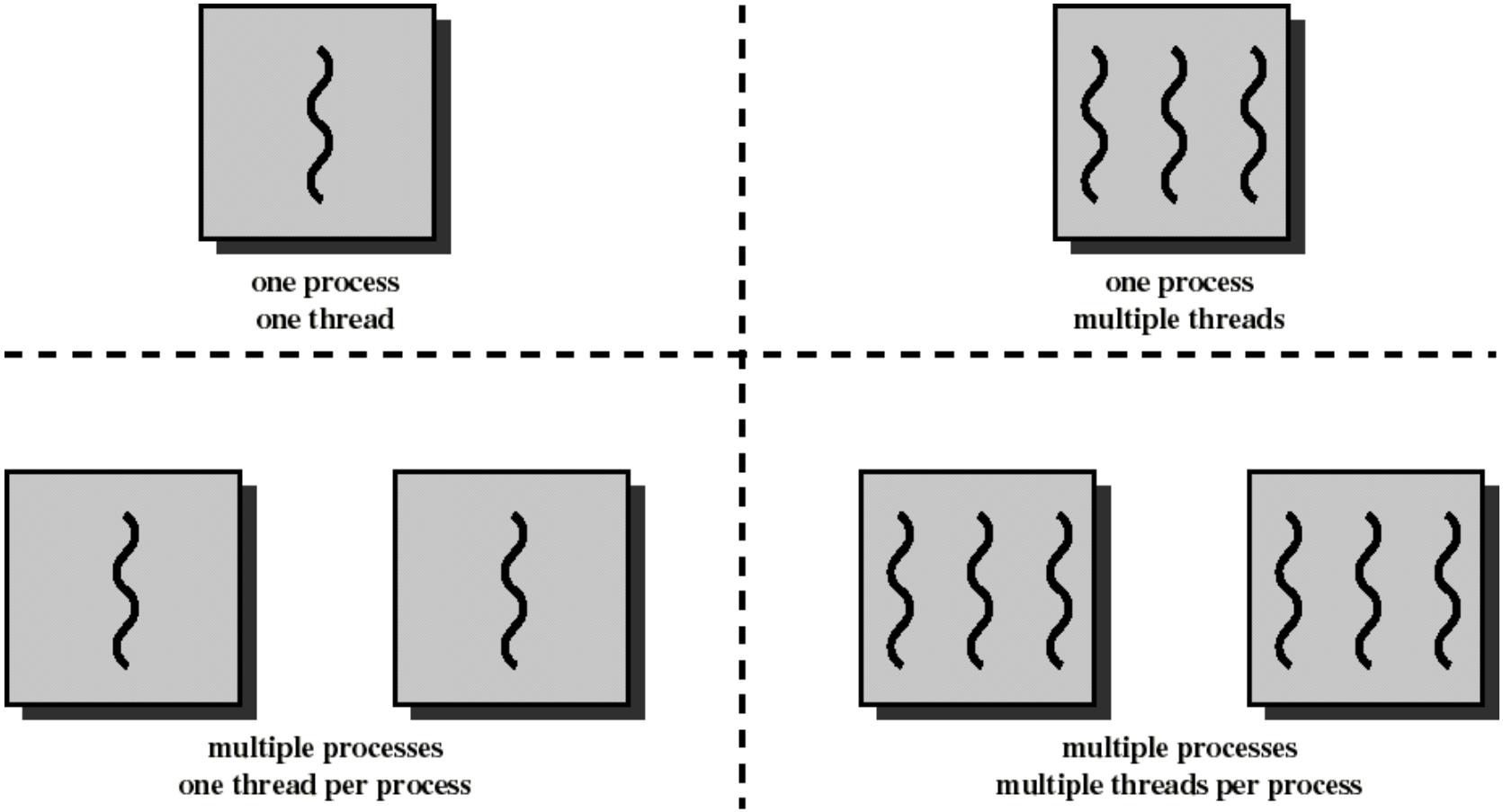
Processus à un thread et à plusieurs threads



Mono-flot

Multi-flots

Threads et processus [Stallings]



Thread

Comme le processus il a un etat comme sont parent(processus)

- **Possède un état d'exécution (prêt, bloqué…)**
- **Possède sa pile et un espace privé pour les variables locales**
- **A accès à l'espace adressable, fichiers et ressources du processus auquel il appartient**
  - En commun avec les autres threads du même processus

Utilité des threads

- **Réactivité: un processus peut être subdivisé en plusieurs threads, ex. l'un dédié à l'interaction avec les usagers, l'autre dédié à traiter des données**
  - L'un peut exécuter tant que l'autre est bloqué

- **Utilisation de multiprocesseurs: les threads peuvent s'exécuter en parallèle sur des UCT différentes**

La commutation entre threads est moins couteuse que la commutation entre processus

- **Un processus possède de la mémoire, des fichiers et d'autres ressources**
- **Changer d`un processus à un autre implique sauvegarder et rétablir l'état de toutes ces ressources**
- **Changer d'un thread à un autre *dans le même processus* est bien plus simple et plus rapide: la sauvegarde des registres de l 'UCT, la pile, et peu d'autres choses**

La communication aussi est moins couteuse entre threads qu'entre processus

- **Étant donné que les threads partagent leur mémoire,**
  - la communication entre threads dans un même processus est plus efficace que la communication entre processus

Efficacite (temps et ressource utiliser)

La création est moins dispendieuse

- **La création et terminaison de nouveaux threads dans un processus existant est aussi moins couteuse que la création d'un processus**

Threads de noyau (kernel) et d'utilisateur

- **Où implémenter les threads:**
  - Dans les bibliothèques de l'usager
    - **contrôlés par l'usager**
    - **POSIX Pthreads, Java threads, Win32 threads**
  - Dans le noyau du SE:
    - **contrôlés par le noyau**
    - **Windows XP/2000, Solaris, Linux, True64 UNIX, Mac OS X**
  - Solutions mixtes
    - **Solaris 2, Windows 2000/NT**

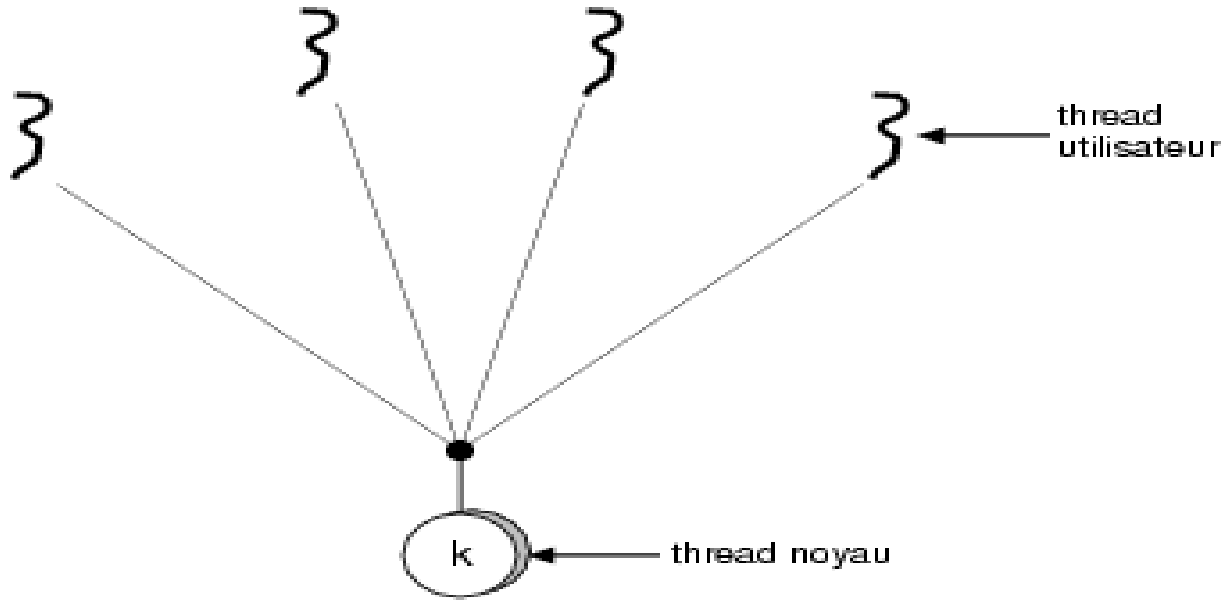
Threads d'utilisateur et de noyau (kernel)

- **Threads d'utilisateur: Supportés par des bibliothèques d'usagers ou de langages de programmation**
  - Efficace car les opérations de threads ne demandent pas d'appels de système
  - Désavantage: le noyau n 'est pas capable de distinguer un état de processus d'un état des threads dans ce processus
    - **Donc le blocage d'un thread implique le blocage du processus**
- **Threads de noyau: Supportés directement par le noyau du SE (WIN NT, Solaris)**
  - Le noyau est capable de gérer directement les états des threads
  - Il peut affecter différents threads à différents UCTs

Solutions mixtes: threads utilisateur et noyau

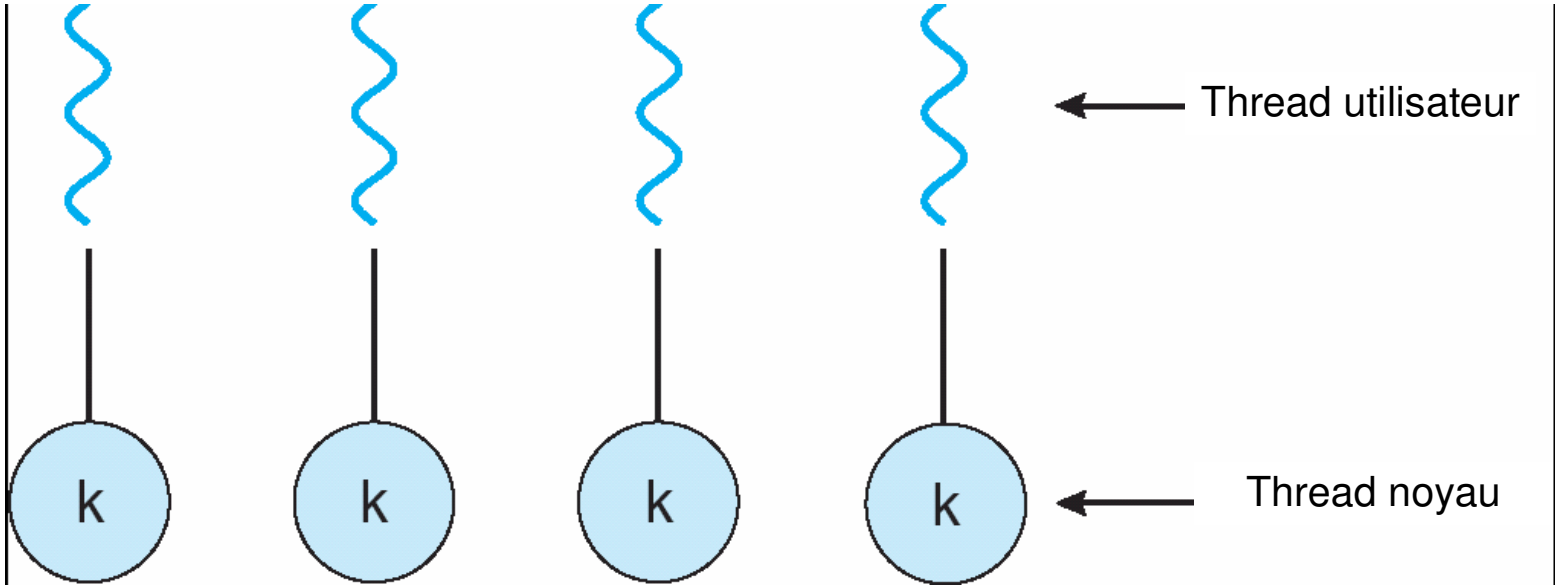
- **Relations entre threads utilisateur et threads noyau**
  - Modèle de plusieurs à un
  - Modèle d'un à un
  - Plusieurs à plusieurs (2 modèles)
- **Nous devons prendre en considération les niveaux suivants:**
  - Processus
  - Thread usager
  - Thread noyau
  - Processeur (UCT)

Plusieurs threads utilisateur pour un thread noyau:
l'usager contrôle les threads



- **Le SE ne connaît pas les threads utilisateur**
  - Avantages et désavantages mentionnés avant
- **Exemples**
  - Solaris Green Threads
  - GNU Portable Threads

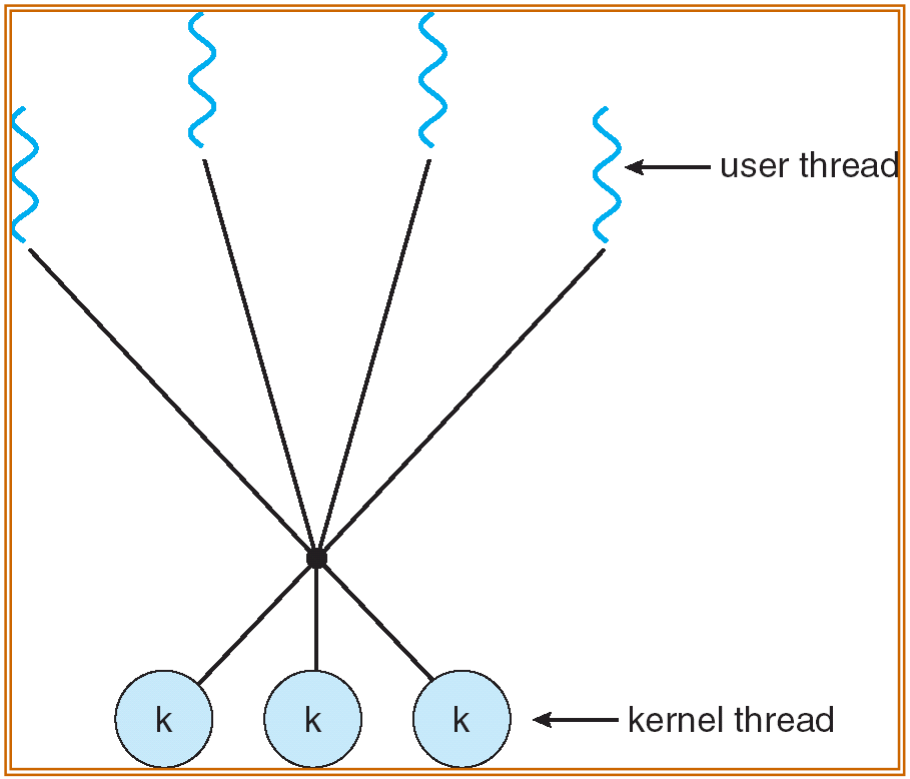
Un à un: le SE contrôle les threads



- **Les opérations de threads sont des appels de système**
- **Permet à un autre thread de s'exécuter lorsqu'un thread exécute un appel de système bloquant**
- **Win NT, XP, OS/2**
- **Linux, Solaris 9**

Plusieurs à plusieurs: solution mixte (M:M – many to many)

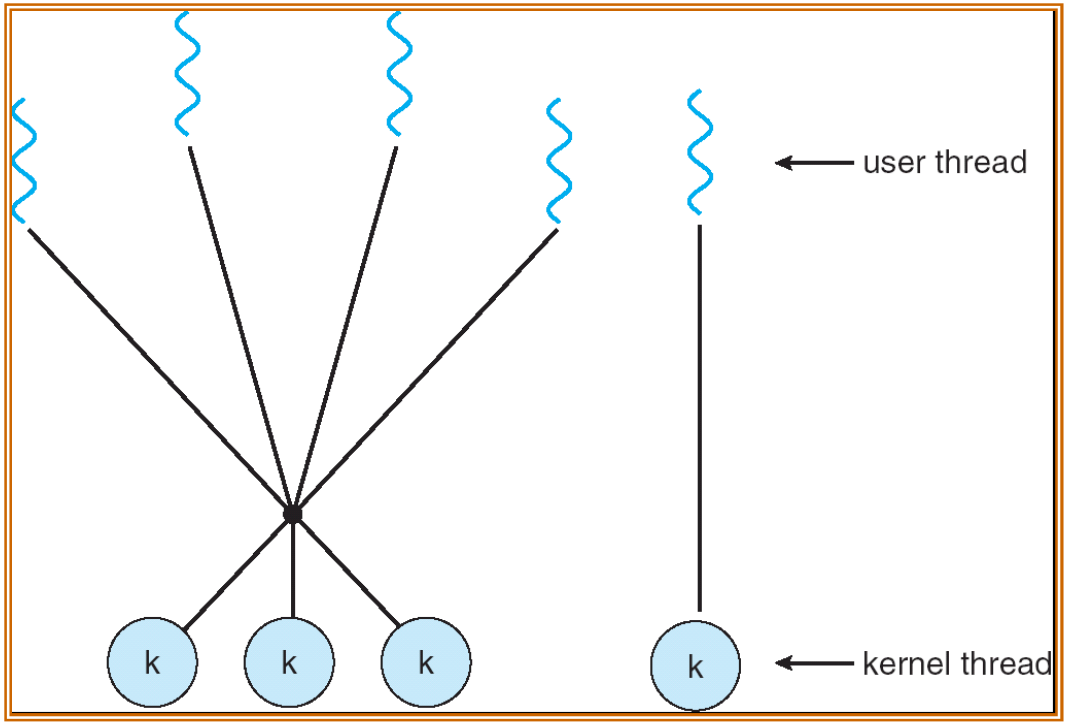
- **Utilise autant de threads utilisateur, que de threads noyau**
- **Flexibilité pour l'utilisateur d'utiliser la technique qu'il préfère**
- **Si un thread utilisateur bloque, son thread kernel peut être affecté à un autre**
- **Si plus d'un UCT est disponible, plusieurs threads kernel peuvent s'exécuter en même temps**
- **Exemples:**
  - Quelques versions d'Unix, dont Solaris avant la version 9
  - Windows NT/2000 avec le *ThreadFiber* package



Modèle à deux niveaux

- **Semblable au M:M, mais permet de rattacher un fil utilisateur à un fil noyau**

- **Exemples**
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 et avant



Multithreads et mono-threads

- **MS-DOS supporte un processus usager à monothread**
- **UNIX SVR4 supporte plusieurs processus à monothread**
- **Solaris, Widows NT, XP et OS2 supportent plusieurs processus multithreads**

Sujets

- **La fil d'exécution chez les processus**
- **Multi-fils versus fil unique (le thread)**
  - Les fils niveau usager et les fils niveau noyau
- **Les défis du « Threading »**
- **Exemples de fils**

Défis du “Threading”

C’est bien beau d’avoir des fils, mais quels sont les conséquences au niveau pratique?

Défis:

- Sémantique des appels systèmes **fork()** et **exec()**
- L’annulation des threads
- Les signaux
- Un groupement de threads (pools)
- Les données spécifiques des threads
- L’ordonnancement

Sémantiques de fork() et exec()

- **Est-ce que fork() copie seulement le fil qui en fait l'appel ou tous les fils?**
  - Souvent les deux versions sont disponibles
  - Lequel utiliser?
- **Qu'est ce qu'exec() fait?**
  - Il remplace l'espace d'adresse, donc tous les fils sont remplacés

L'annulation du thread (cancellation)

- **Terminaison du thread avant qu'il ne soit fini.**
- **Deux approches générales:**
  - **Annulation asynchrone** qui termine le fils immédiatement
    - **Peut laisser les données partagées dans un état corrompu** On sait pas ds quel etat se trouve les donnees
    - **Certaines ressources peuvent ne pas être libérées**.
  - **Annulation différé** Souvent representer comme un ralentissement
    - **Utilise un drapeau que le fils vérifie pour savoir s'il doit annuler son exécution**
    - **Permet une terminaison plus en douceur**

      Plus facile pour le debuggage

Les signaux

- **Les signaux sont utilisés en UNIX pour aviser un processus de l'arrivée d'un évènement**
- **Essentiellement une interruption de logiciel**
- **Un gestionnaire de signaux traite les signaux**
  - Un signal est créé par un évènement particulier
  - Le signal est délivré à un processus
  - Le signal est traité
- **Options:**
  - Livraison du signal au fil correspondant
  - Livraison du signal à tous les fils du processus
  - Livraison du signal à certains fils du processus
  - Attribue à un fil spécifique la réception des signaux et de leurs traitements

Les regroupements de fils (Thread Pools)

- **Un processus serveur peut desservir ses requêtes en créant un fil pour chacune de celle qui en a besoin**
  - La création de fil est couteuse et prend du temps
  - Il n'y a pas de contrôle sur le nombre de fils, ce qui peut surcharger le système.
- **Solution**
  - Créer un certain nombre de fils qui attendent du travail
  - Avantages:
    - **Le temps de création n'as lieu qu'au début de la création du groupe de fils**
    - **Le nombre de fils exécutant est limité par la taille du groupe**

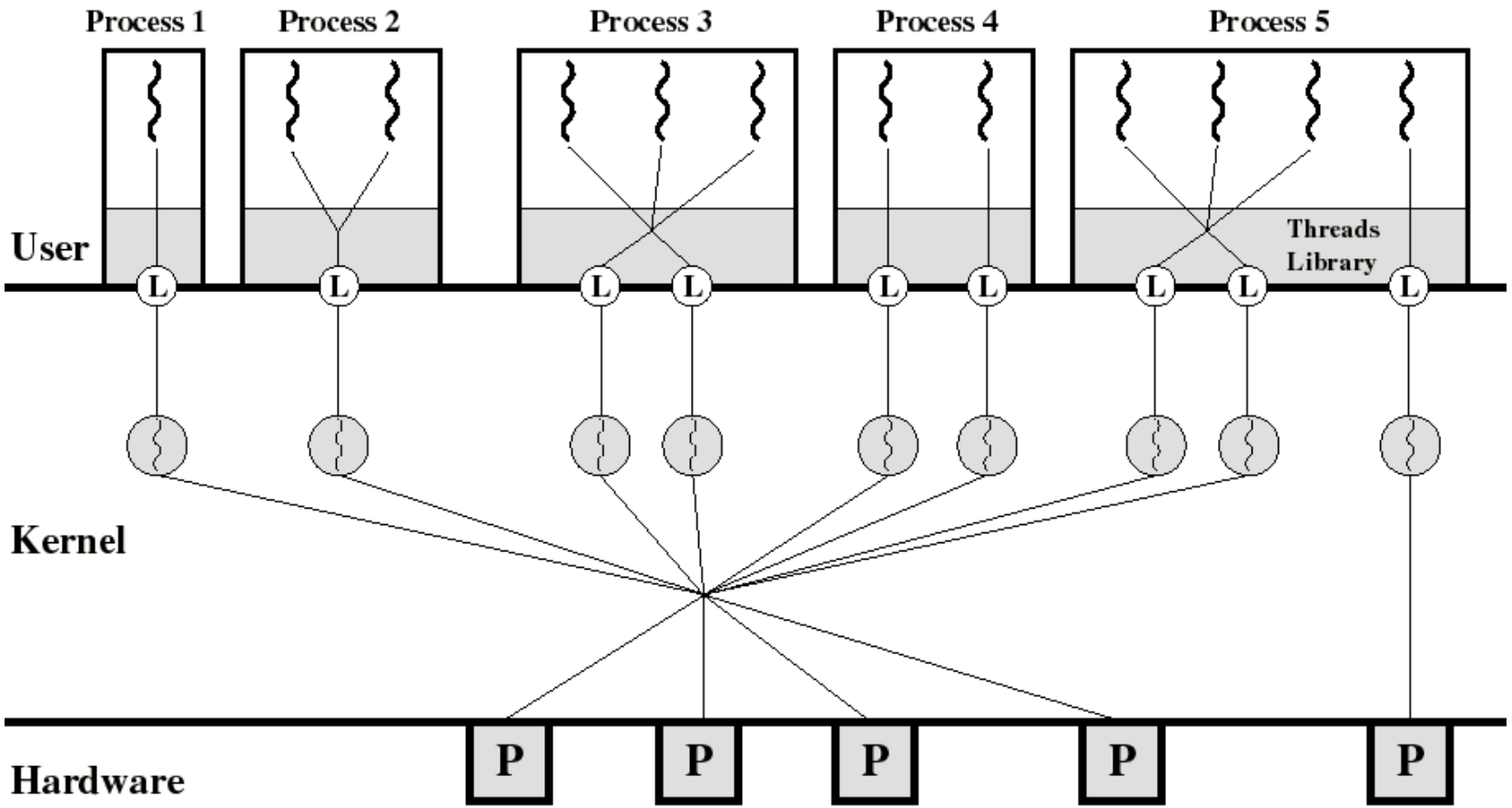
Avantage : On a pas a creer de thread a la demande. On en cree au depart et ensuite des qu'un thread finit il en a un qui attend pour lui

Données spécifiques aux fils

- **Permet à chaque fil d'avoir sa propre copie de données**
- **Pratique lorsque le contrôle de la création du fil est limité (i.e. dans un groupe de fils).**

La planification (scheduler activations)

- **La communication à partir du noyau est nécessaire pour informer la bibliothèque de fils lorsque le fil usager s'apprête à être bloqué et lorsqu'il redevient prêt à l'exécution.**
  - Modèle plusieurs à plusieurs
  - Modèle à deux niveaux
- **Lorsque ces évènements se produisent, le noyau fait un appel vers l'amont (upcall) de la bibliothèque de fils**
- **Le gestionnaire des upcall de la bibliothèque de fils gère cet évènement (c.à.d sauvegarde l'état du fil et le change à un état bloqué).**



Processus 2 est équivalent à une approche ULT

Processus 4 est équivalent à une approche KLT

Nous pouvons ajuster le degré de parallélisme du noyau (processus 3 et 5)

Stallings

Sujets

- **La fil d'exécution chez les processus**
- **Multi-fils versus fil unique (le thread)**
  - Les fils niveau usager et les fils niveau noyau
- **Les défis du « Threading »**
- **Exemples de fils**

Exemples de bibliothèques de fil

- **Pthreads**
- **Win32**
- **Java threads**

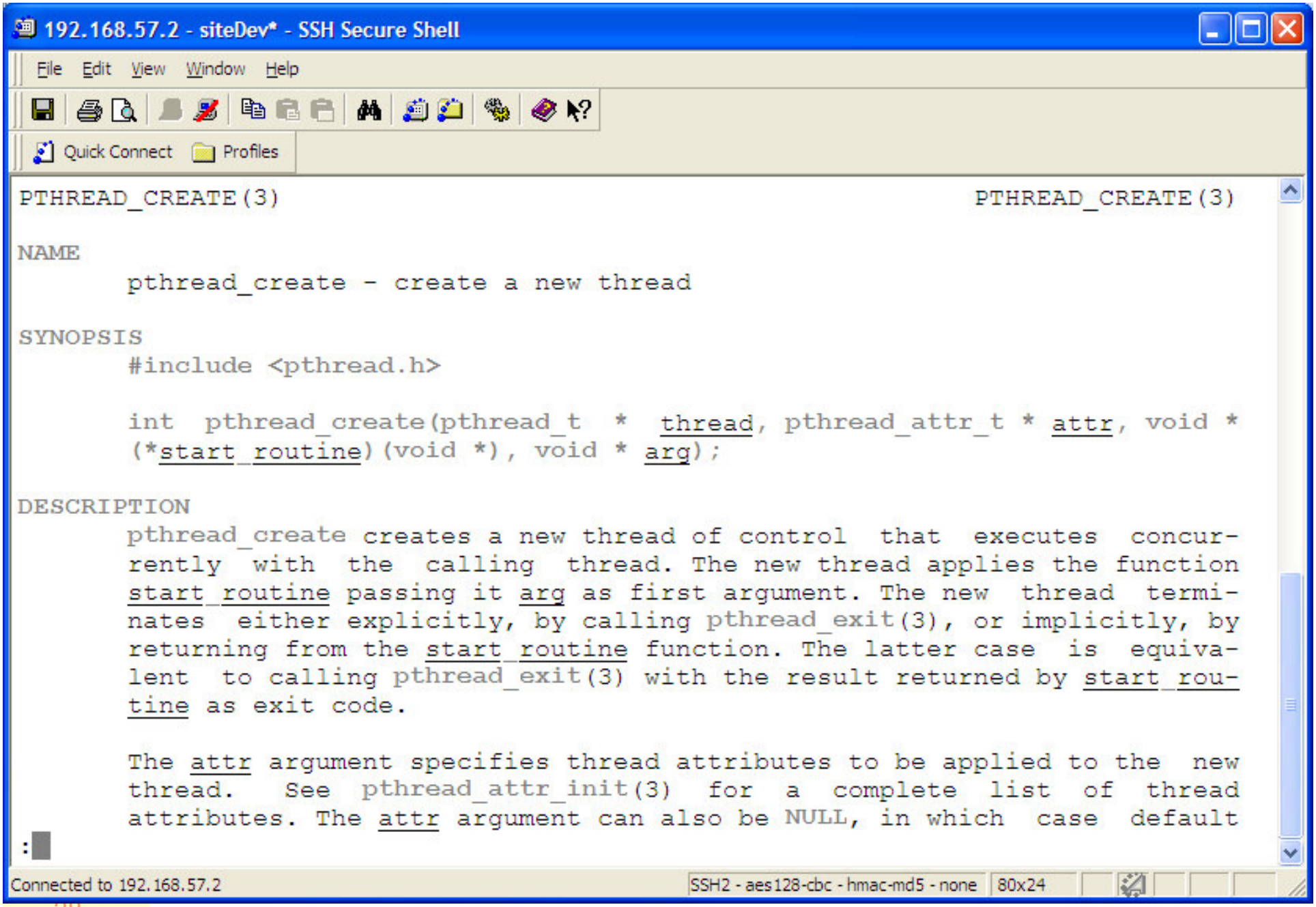
Pthreads

- **C'est une norme POSIX (IEEE 1003.1c) d'API pour la création et synchronisation de fils**
- **L'API spécifie le comportement de la bibliothèque de fils (sa réalisation dépend du développeur)**
- **Commun dans les systèmes d'exploitations UNIX (Solaris, Linux, Mac OS X)**
- **Fonctions typiques:** Parametre pour create : Ce sont tous des pointeurs

Creation

  - pthread_create (&threadid,&attr,start_routine,arg)
  - pthread_exit (status)
  - pthread_join (threadid,status)
  - pthread_attr_init (&attr)

Methode quil va rouler


192.168.57.2 - siteDev* - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
PTHREAD_CREATE(3) PTHREAD_CREATE(3)
NAME
pthread_create - create a new thread
SYNOPSIS
#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t * attr, void *
(*start_routine)(void *), void * arg);
DESCRIPTION
pthread_create creates a new thread of control that executes concur-
rently with the calling thread. The new thread applies the function
start_routine passing it arg as first argument. The new thread termi-
nates either explicitly, by calling pthread_exit(3), or implicitly, by
returning from the start_routine function. The latter case is equiva-
lent to calling pthread_exit(3) with the result returned by start_rou-
tine as exit code.
The attr argument specifies thread attributes to be applied to the new
thread. See pthread_attr_init(3) for a complete list of thread
attributes. The attr argument can also be NULL, in which case default
:
Connected to 192.168.57.2
SSH2 - aes128-cbc - hmac-md5 - none
80x24

Exercice de programmation avec fils

Objectif: Écrire un programme de multiplication de matrice avec plusieurs fils, dans le but de profiter de plusieurs UCTs.

Programme pour la multiplication, avec mono-fil, des matrices A et B d'ordre n x n

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++) {
        C[i,j] = 0;
        for(k=0; k<n; k++)
            C[i,j] += A[i,k] * B[k,j];
    }
```

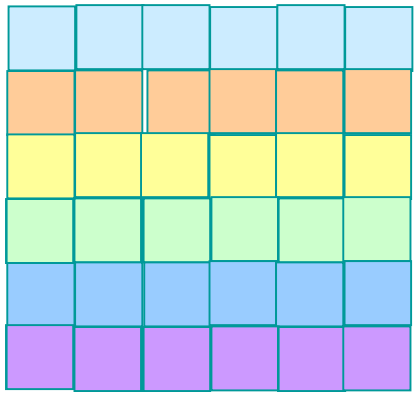
Pour rendre notre tache plus facile:

On assume qu'il y a 6 UCTs et que **n** est un multiple de 6

La multiplication de matrice avec multi-fils

Idée:

- création de 6 fils
- chaque fil résout 1/6 de la matrice C
- attendons que les 6 fils finissent
- la matrice C peut maintenant être utilisée



Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

Allons-y!

```
pthread_t tid[6];
pthread_attr_t attr;
int i;

pthread_init_attr(&attr);
for(i=0; i<6; i++)   /* création des fils de travail */
   pthread_create( &tid[i], &attr, travailleur, &i);

for(i=0; i<6; i++)  /* attendons que tous soient fini */
   pthread_join(tid[i], NULL);

/* la matrice C peut maintenant être utilisée */
…
   /* la variable i est transferée aux fils mais pas sa
   valeur */
```

Allons-y!

```
void *travailleur(void *param)
{
    int i,j,k;
    int id = *((int *) param); /* interprétons param
                                  come un pointeur d'un entier
    */
    int bas = id*n/6;
    int haut = (id+1)*n/6;

    for(i=bas; i<haut; i++)
       for(j=0; j<n; j++)
       {
          C[i,j] = 0;
          for(k=0; k<n; k++)
             C[i,j] = A[i,k]*B[k,j];
       }
    pthread_exit(0);
}
```

Allons-y!

Cela fonctionne-t-il?

- A-t-on besoin de passer A, B, C et n comme paramètres?
  - non, ils sont dans la mémoire partagée, on est ok
- Est ce que les IDs ont bien été passés?
  - pas vraiment, les pointeurs reçoivent tous la même adresse.

```
int id[6];
.
.
for(i=0; i<6; i++)    /* create the working threads */
{
    id[i] = i;
    pthread_create( &tid[i], &attr, worker, &id[i]);
}
```

Cela fonctionne-t-il maintenant?

- Cela devrait, … &id[i] est une variable locale pour chaque fil

API du thread Win32

```
// création d'un thread
ThreadHandle = CreateThread(
    NULL,                   // attributs de sécurité par défaut
    0,                      // taille de la pile par défaut
    Summation,              // fonction à éxecuter
    &Param,                 // paramètre de la fonction thread
    0,                      // création des fanions par défaut
    &ThreadId);             // retourn l'ID du thread

if (ThreadHandle != NULL) {
    WaitForSingleObject(ThreadHandle, INFINITE);
    CloseHandle(ThreadHandle);
    printf("sum = %d\n",Sum);
}
```

Threads Java

- **Les fils Java sont crées avec un appel à la méthode start() d'une classe qui :**
  - Étend (extend) la classe Thread, ou
  - Utilise l'interface Runnable:

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

- **Les fils Java sont héritables et représentent une partie importante du langage Java qui offre un éventail d'API riches de fonctions.**

Extension de la classe Thread

```
class Worker1 extends Thread
{
                    "Run" Doit etre utiliser pour pour faire rouler le thread
  public void run() {
    System.out.println(“Je suis un fil au travail");
  }
}

public class First
{
  public static void main(String args[]) {
    Worker1 runner = new Worker1();
    runner.start();
    System.out.println("Je suis le fil principal");
  }
}
```

Implémentation de l'interface Runnable

```
class Worker2 implements Runnable
{
   public void run() {
      System.out.println(" Je suis un fil au travail");
   }
}
public class Second
{
   public static void main(String args[]) {
      Runnable runner = new Worker2();
      Thread thrd = new Thread(runner);
      thrd.start();

      System.out.println("Je suis le fil principal");
   }
}
/* Worker1 et 2 sont deux programmes qui réalisent la même chose */
```

Jointure de Threads

```
class JoinableWorker implements Runnable
{
  public void run() {
    System.out.println("Travaileur au travail");
  }
}

public class JoinExample
{
  public static void main(String[] args) {
    Thread task = new Thread(new JoinableWorker());
    task.start();

    try { task.join(); }
    catch (InterruptedException ie) { }

    System.out.println("le travaileur a fini");
  }
}
```

Annulation de Thread

```
Thread thrd = new Thread (new InterruptibleThread());
Thrd.start();

. . .

// interrompons le maintenant
Thrd.interrupt();
```

Annulation de Thread

```
public class InterruptibleThread implements Runnable
{
   public void run() {
      while (true) {
         /**
          * Effectue d'autres travaux en attendant
          */

         if (Thread.currentThread().isInterrupted()) {
            System.out.println("J'ai été interrompu!");
            break;
         }
      }

      // nettoyage et terminaison
   }
}
```

Données spécifiques de Thread

```
class Service
{
  private static ThreadLocal errorCode = new ThreadLocal();

  public static void transaction() {
    try {
      /**
       * une opération où une erreur peut se produire
       */
    catch (Exception e) {
      errorCode.set(e);
    }
  }

  /**
   * récupération du code d'erreur de la transaction
   */
  public static Object getErrorCode() {
    return errorCode.get();
  }
}
```

Données spécifiques de Thread

```
class Worker implements Runnable
{
   private static Service provider;

   public void run() {

     provider.transaction();
      System.out.println(provider.getErrorCode());
   }
}
```

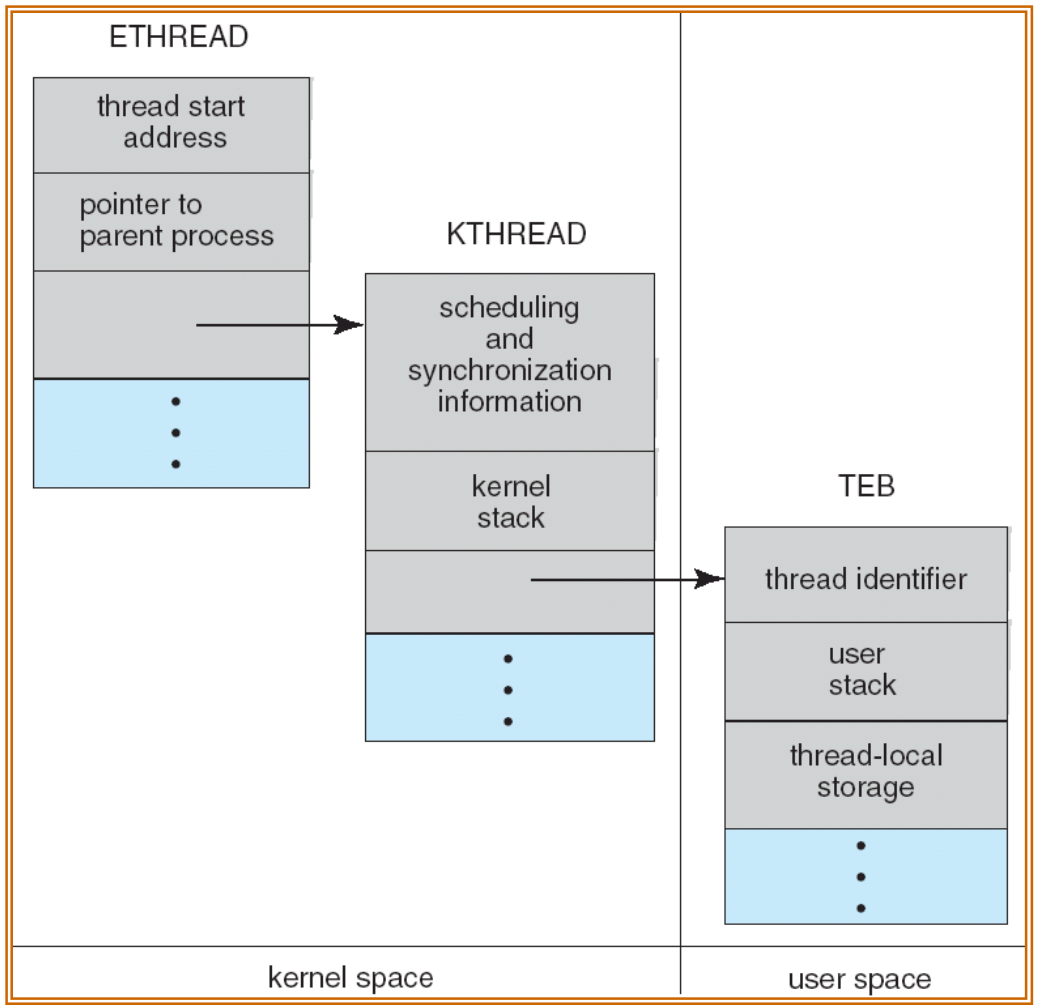
Exemples d'Implémentation de fil chez les E/S

- **Windows XP**
- **Linux**
- **Java**

Threads de Windows XP

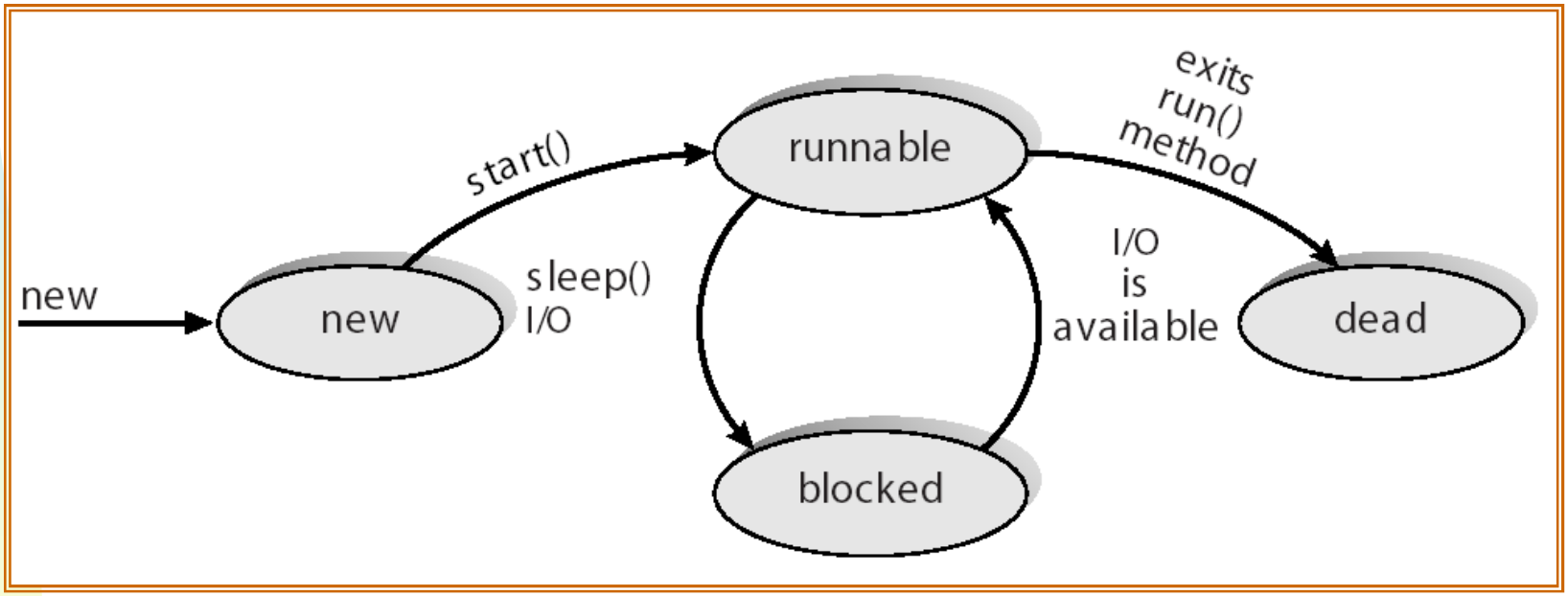
- **Modèle un à un**
- **Chaque fil contient**
  - Un identificateur de fil (id)
  - Un ensemble de registres
  - Différentes piles d'utilisateur et de noyau
  - Une mémoire privée de données
- **L'ensemble des registres, piles et de la mémoire privée forme le *contexte* du fil**
- **Les structures principales de données d'un fil comprennent:**
  - ETHREAD (bloc de fils éxécutoires)
  - KTHREAD (bloc de fils kernel)
  - TEB (bloc de fils environnementaux)

Threads de Windows XP



Les fils Java

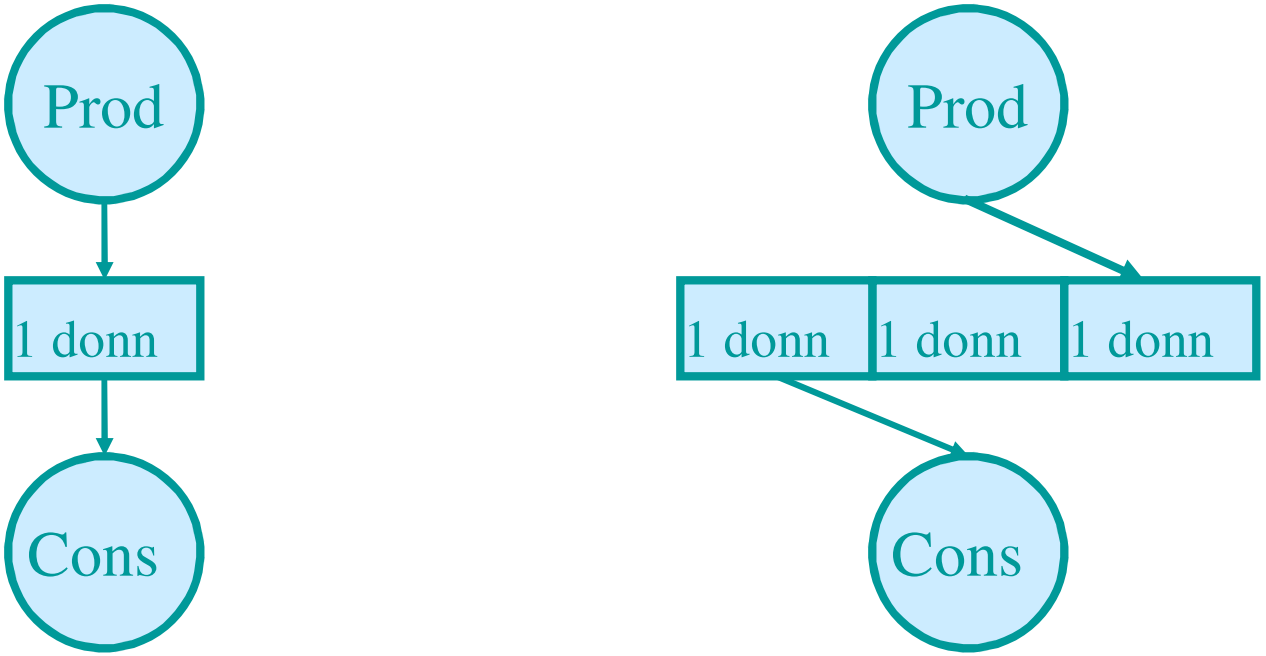
Les fils Java sont gérés par le JVM



Problème producteur - consommateur

- **Un problème classique dans l'étude des processus communicants**
  - un processus *producteur* produit des données (ex. les enregistrements d'un fichier) pour un processus *consommateur*
  - un programme d'impression produit des caractères -- consommés par une imprimante
  - un assembleur produit des modules objets qui seront consommés par le chargeur
- **Nécessité d'un tampon pour stocker les items produits (attendant d'être consommés)**

Tampons de communication

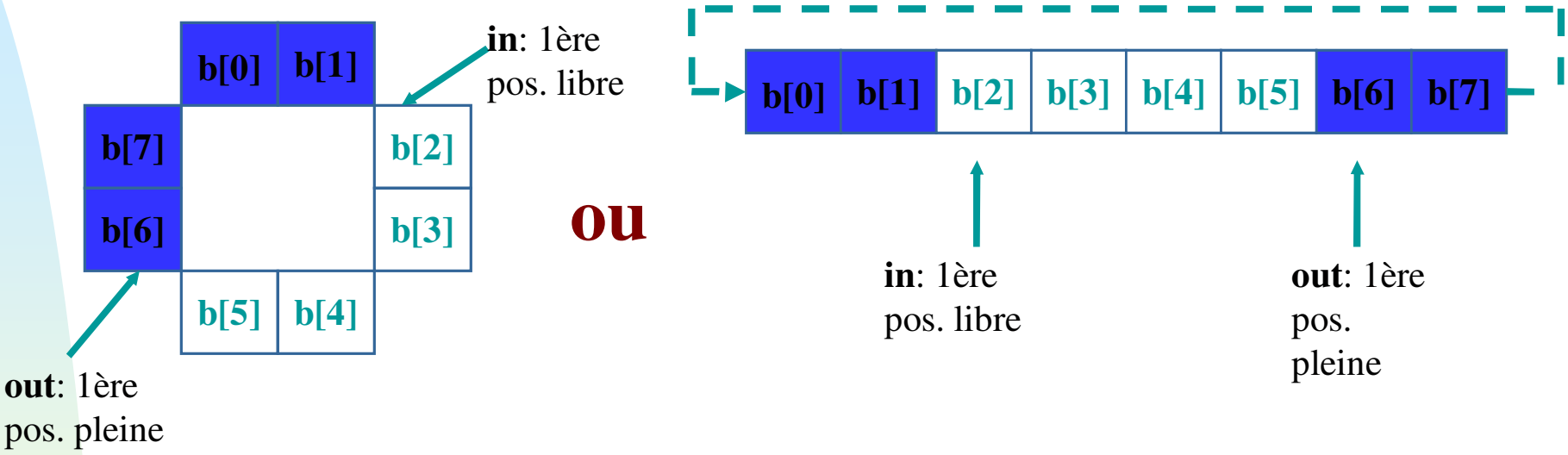


Si le tampon est de longueur 1, le producteur et consommateur doivent forcement aller à la même vitesse

Des tampons de longueur plus grandes permettent une certaine indépendance. ex. à droite le consommateur a été plus lent

Le producteur peut ajouter juska la longueur du tampon et en meme temps le consommateur fait sont travail.
Meme s'il est plus lent le producteur peut ecrire dans les espaces libres du tampon. Les deux vont a leur vitesse

Le tampon borné (bounded buffer)
une structure de données fondamentale dans les SE



Le tampon borné se trouve dans la mémoire partagée entre le consommateur et l'usager

bleu: plein, blanc: libre

Problème producteur - consommateur

```
public class Factory
{
   public Factory()   {
      // créons d'abord le tampon du message
      MessageQueue mailBox = new MessageQueue();

      // now créons maintenant les fils producteur et consommateur
      Thread producerThread = new Thread(new Producer(mailBox));
      Thread consumerThread = new Thread(new Consumer(mailBox));

      producerThread.start();
      consumerThread.start();
   }

   public static void main(String args[]) {
      Factory server = new Factory();
   }
}
```

Fil producteur

```
class Producer implements Runnable
{
   private  MessageQueue mbox;

   public Producer(MessageQueue mbox) {
      this.mbox = mbox;
   }

   public void run() {
      Date message;

      while (true) {
         SleepUtilities.nap();
         message = new Date();
         System.out.println("Le producteur produit " + message);

         // production d'un item  et son stockage dans le tampon
         mbox.send(message);
      }
   }
}
```

Fil consommateur

```
class Consumer implements Runnable
{
  private  MessageQueue mbox;

  public Consumer(MessageQueue mbox) {
    this.mbox = mbox;
  }

  public void run() {
    Date message;

    while (true) {
      SleepUtilities.nap();
      // consommation d'un item du tampon
      System.out.println("Le consommateur veut consommer.");

      message = (Date)mbox.receive();
      if (message != null)
        System.out.println(" Le consommateur a consommé " + message);
    }
  }
}
```