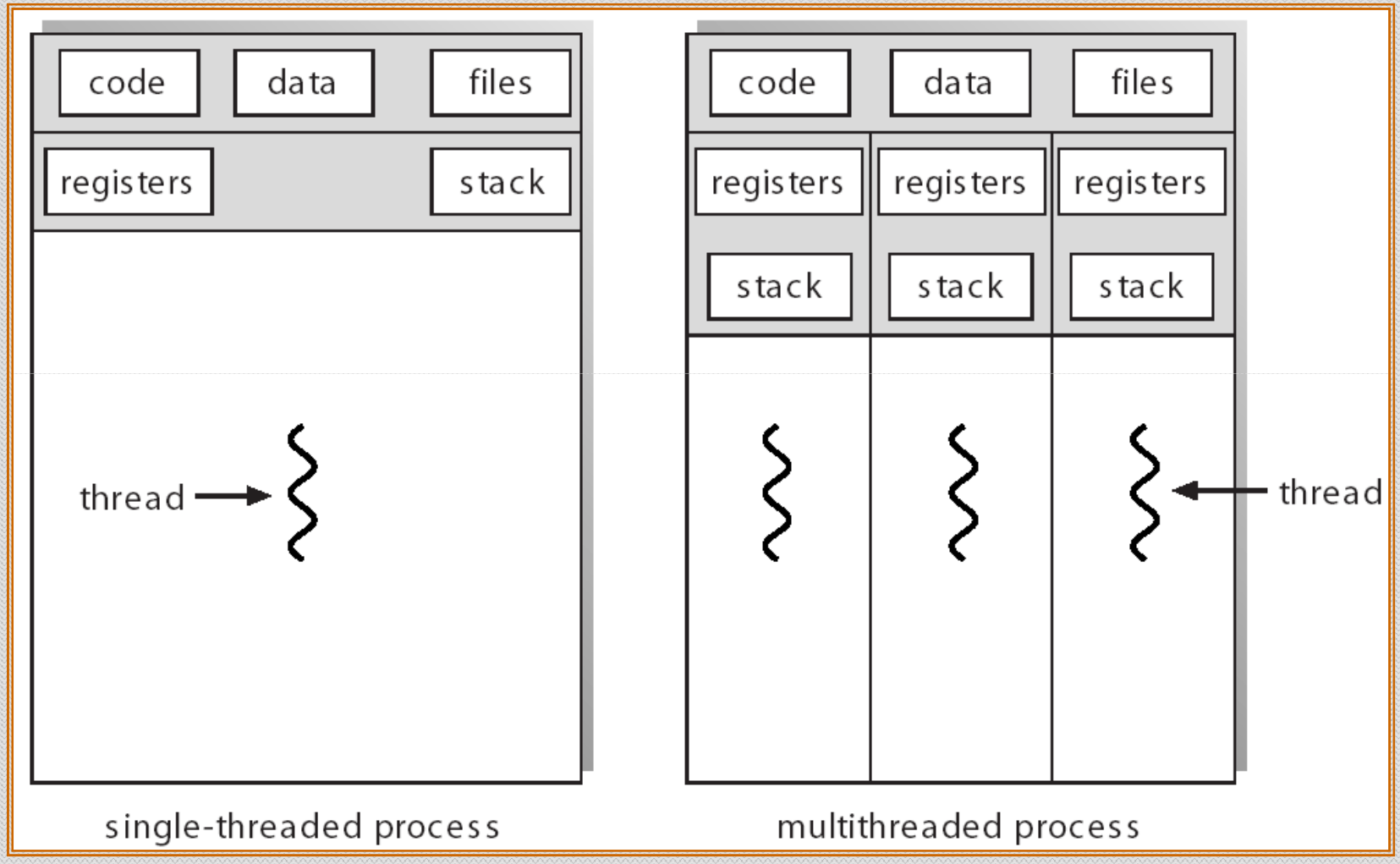
Chapitre 3
Gestion des threads

Support du cours : www.achrafothman.net

Threads

- Vue d'Ensemble
- Modèles de Multithreading
- Problèmes des Threads
- Pthreads
- Threads Windows XP
- Threads Linux
- Threads Java

Processus à Un ou Plusieurs Threads



Avantages

- Réactivité
- Partage de Ressources
- Economie
- Utilisation d'Architectures MP

Threads Utilisateur

- Gestion des threads faite par la bibliothèque au niveau utilisateur

- Trois bibliothèques de threads assez utilisées:
  - Pthreads
  - Threads Java
  - Threads Win32

Threads Noyau

- Supportés par le noyau
- Exemples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Modèles de Multithreading

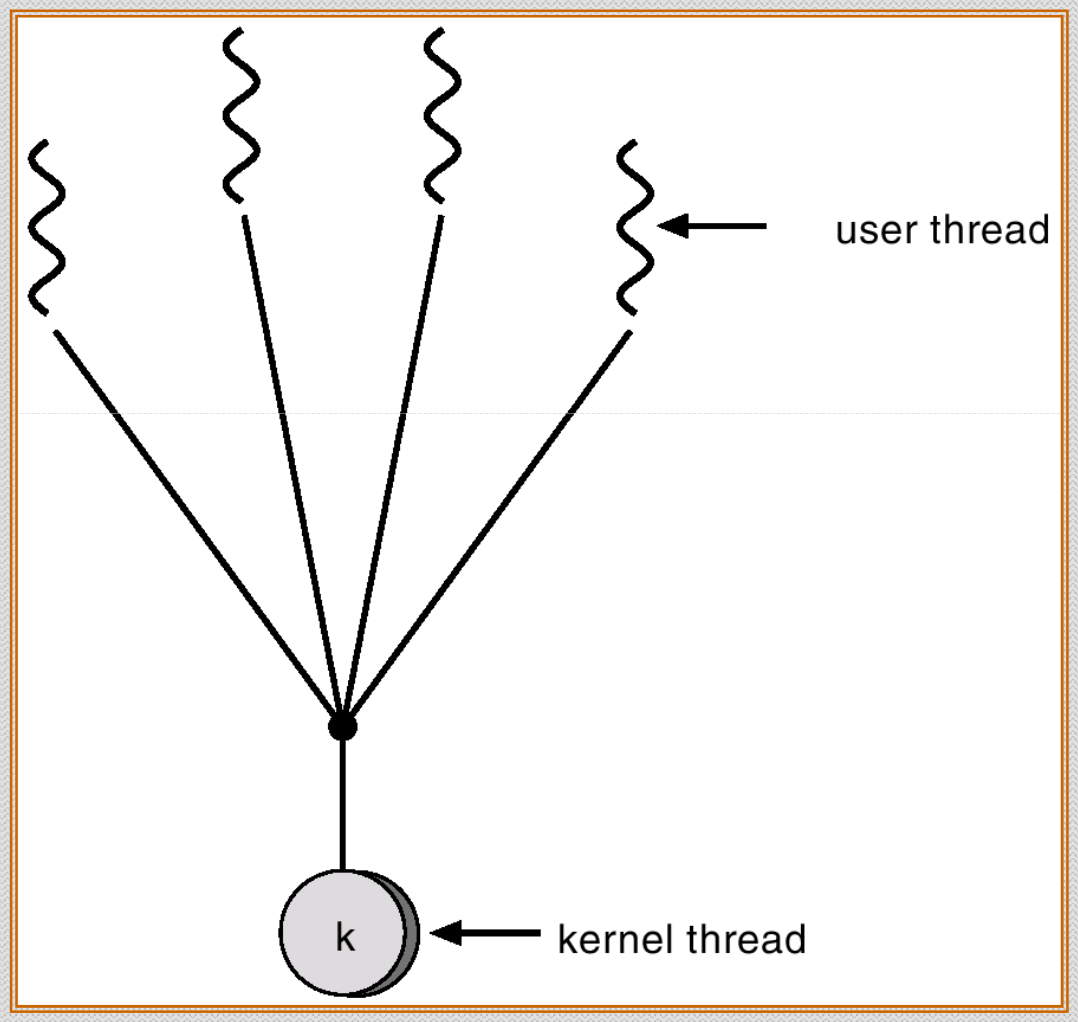
- Plusieurs-à-Un
- Un-à-Un
- Plusieurs-à-Plusieurs

Plusieurs-à-Un

- Plusieurs threads utlisateurs attachés à un seul thread noyau

- Exemples
  - Solaris Green Threads
  - GNU Portable Threads

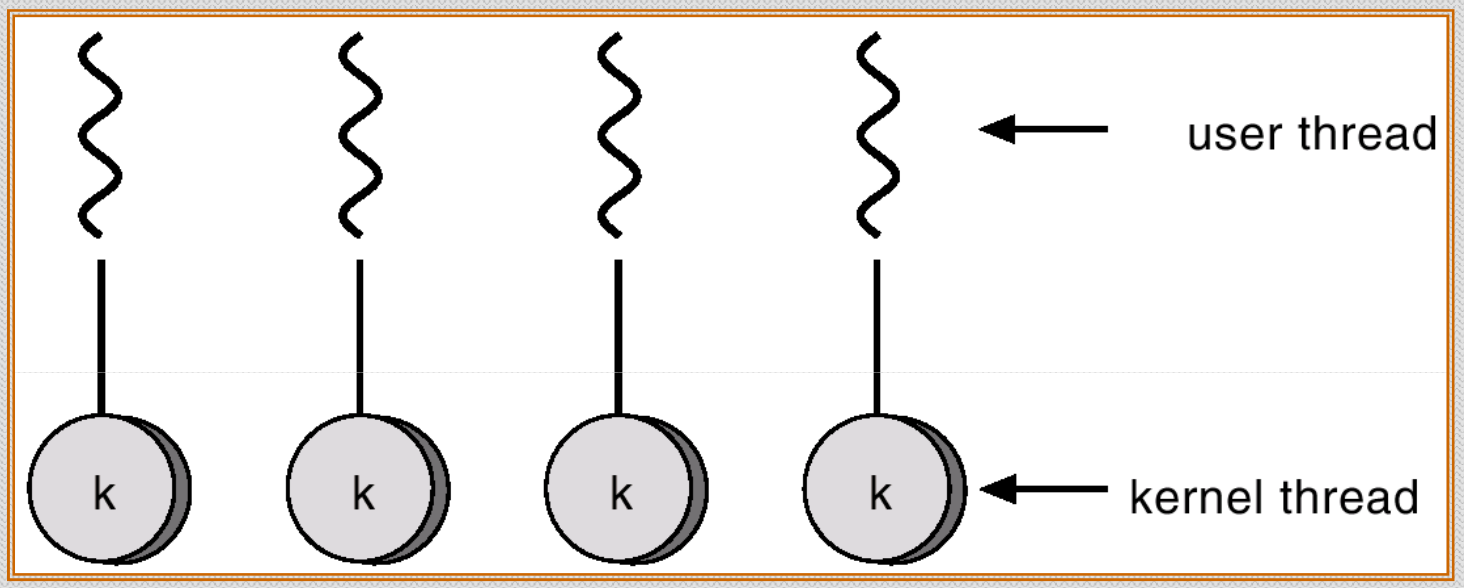
Modèle Plusieurs-à-Un



Modèle Un-à-Un

- Chaque thread utilisateur est attaché à un thread noyau
- Exemples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 et +

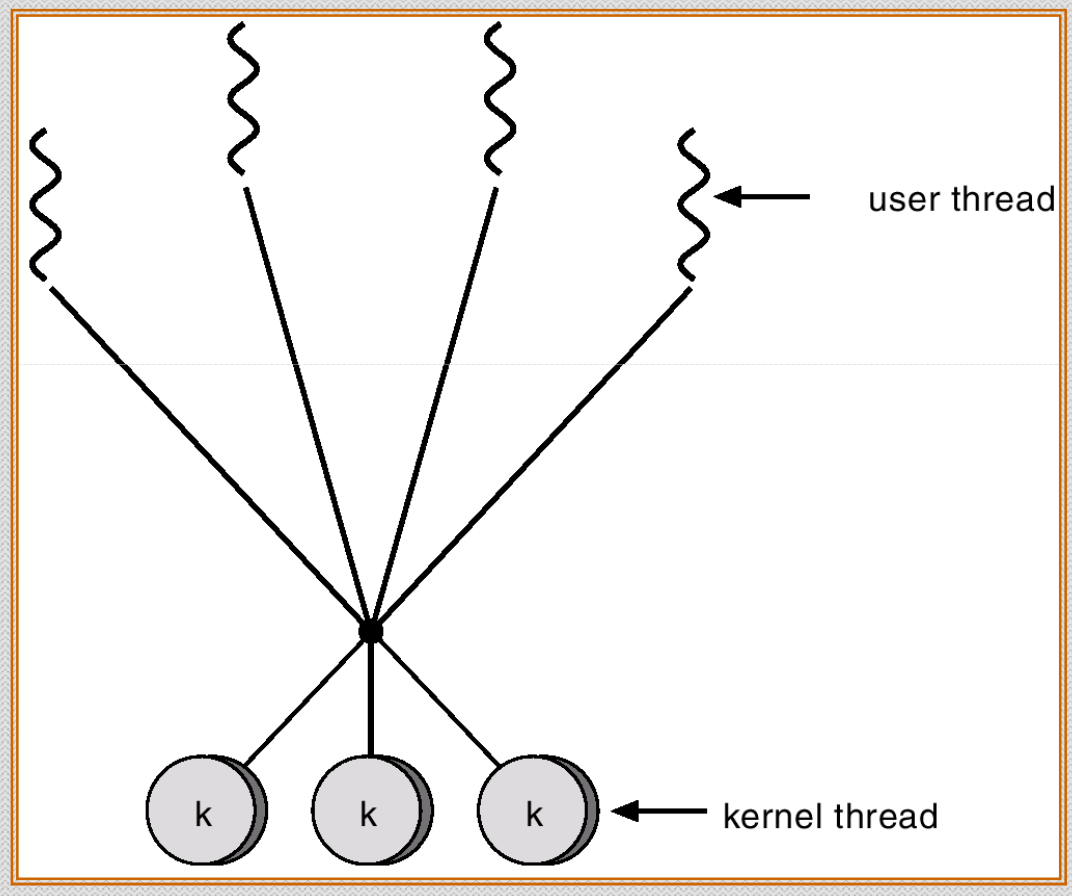
Modèle Un-à-Un



Modèle Plusieurs-à-Plusieurs

- Permet à plusieurs threads utilisateur d'être attachés à plusieurs threads noyaux
- Permet à l'OS de créer un nombre suffisant de threads noyau
- Version de Solaris < 9
- Windows NT/2000 avec le package *ThreadFiber*

Modèle Plusieurs-à-Plusieurs

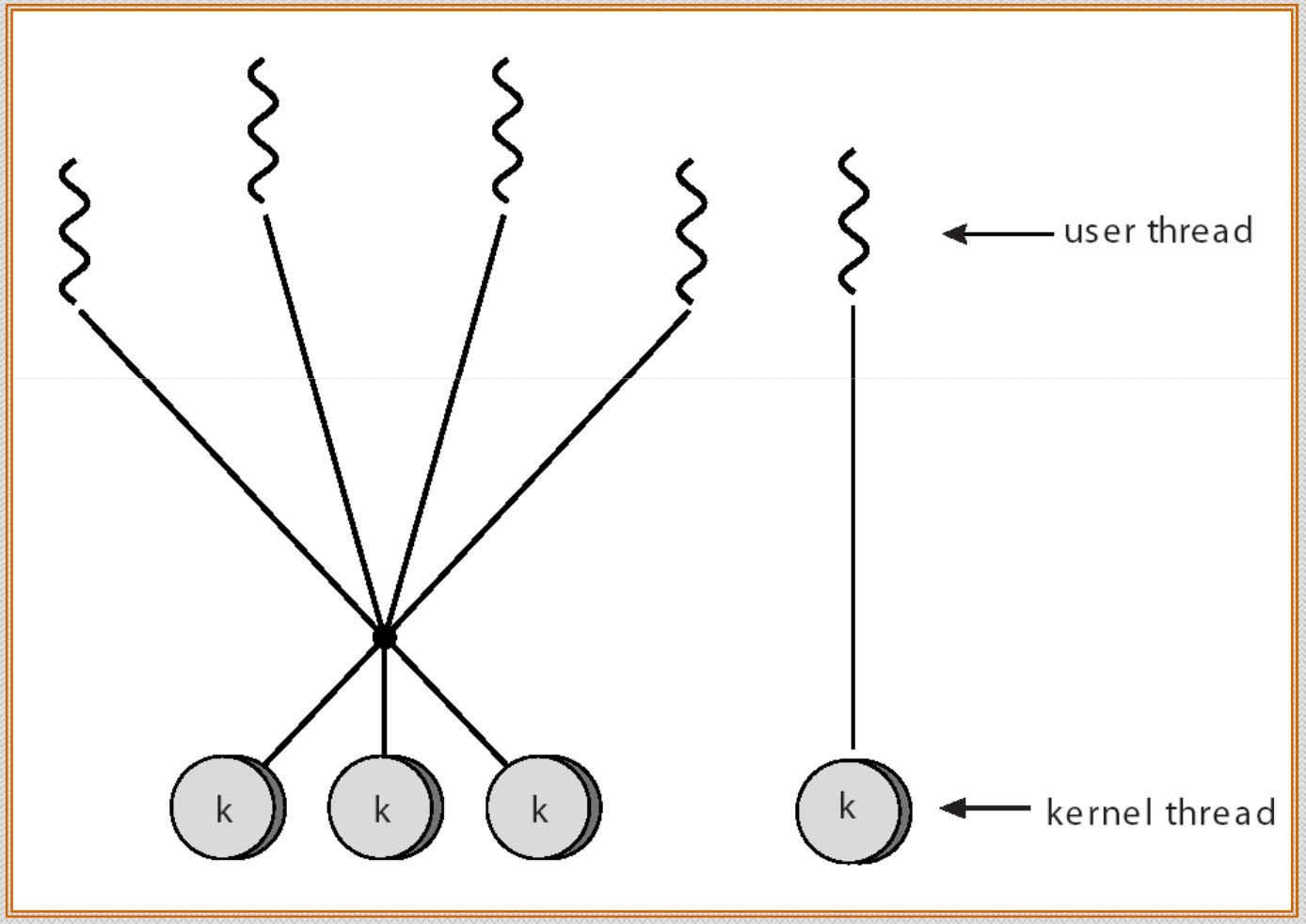


Modèle à Deux Niveaux

- Similaire à P:P, sauf qu'il permet à un thread utilisateur d'être **lié** à un thread noyau

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

Modèle à Deux Niveaux



Questions du Threading

- Sémantiques des appels système **fork()** et **exec()**
- Suppression d'un thread
- Gestion des signaux
- Pools de threads
- Données spécifiques à un thread
- Activations de l'ordonnanceur

Sémantiques de fork() et exec()

- Est-ce que **fork()** duplique seulement le thread appelant ou tous les threads ?

Suppression d'un Thread

- Terminaison d'un thread avant sa fin

- Deux approches générales:
  - **Terminaison asynchrone** termine le thread immédiatement
  - **Terminaison différée** permet au thread de vérifier périodiquement s'il devrait être supprimé

Gestion des Signaux

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific threa to receive all signals for the process

Thread Pools

- Créer un nombre de threads dans un pool où ils attendent d'être utilisés

- Avantages:
  - Il est plus rapide de répondre à une requête avec un thread existant qu'en créant un nouveau
  - Permet au nombre de threads dans une application de se limiter à la taille du pool

Données Spécifiques à un Thread

- Permet à chaque thread d'avoir sa propre copie de données
- Utile quand on n'a pas le contrôle sur la création d'un thread (i.e., en utilisant un pool de threads)

Activation de l'Ordonnanceur

- Les modèles P:P et à deux niveaux requièrent le maintien du nombre approprié de threads noyau alloués à l'application
- Les activations de l'ordonnanceur fournissent des **upcalls** – un mécanisme de communication du noyau vers la bibliothèque de threads
- Cette communication permet à l'application le nombre correct de threads noyau

Pthreads

- Un standard POSIX (IEEE 1003.1c) une API pour la création et la synchronisation des threads
- L'API spécifie le comportement d'une bibliothèque de threads, l'implémentation étant laissée aux développeurs
- Utilisée dans différents OSs UNIX (Solaris, Linux, Mac OS X)

Pthreads

```
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of attributes for the thread */
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* now wait for the thread to exit */
   pthread_join(tid,NULL);
   printf("sum = %d\n",sum);
}

void *runner(void *param) {
   int upper = atoi(param);
   int i;
   sum = 0;
   if (upper > 0) {
      for (i = 1; i <= upper; i++)
         sum += i;}

pthread_exit(0);}
```

Threads Windows

- Implémente le modèle Un-à-Un
- Chaque thread comporte
  - Un id
  - Ensemble de registres
  - Piles utilisateur et noyau séparés
  - Espace de stockage de données séparé
- L'ensemble de registres, piles, et l'espace de stockage privé constituent **contexte** d'un thread
- Les structures de données d'un thread sont:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Threads Linux

- Linux parle plutôt de tâches
- La création de threads se fait à l'aide de l'appel système **clone()**
- **clone()** permet à une tâche fils de partager l'espace d'adressage de la tâche parent (processus)

Threads Threads

- Les threads Java sont gérés par la JVM
- Les threads Java peuvent être créés par:
  - Extension de la classe Thread
  - Implémentation de l'interface Runnable

Extension de la Classe Thread

```
class Worker1 extends Thread
{
   public void run() {
      System.out.println("I Am a Worker Thread");
   }
}

public class First
{
   public static void main(String args[]) {
      Worker1 runner = new Worker1();
      runner.start();

      System.out.println("I Am The Main Thread");
   }
}
```

L'Interface Runnable

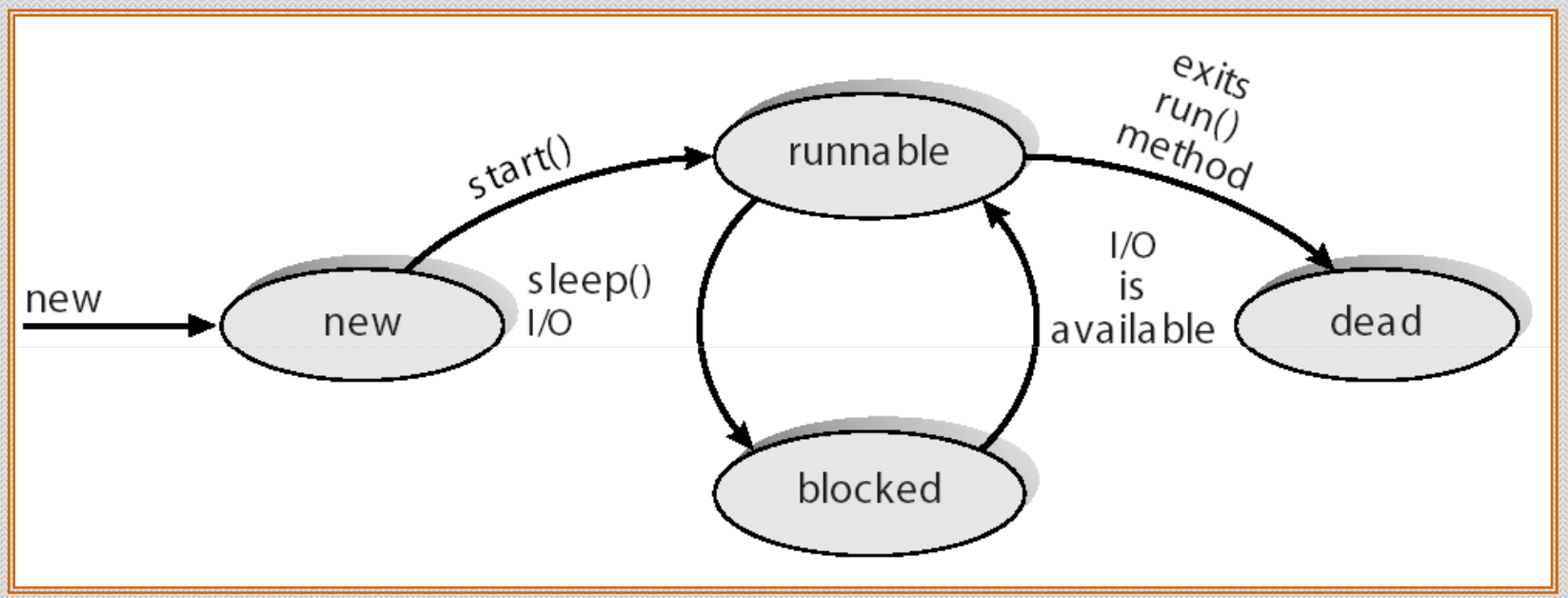
```
public interface Runnable
{
   public abstract void run();
}
```

Implementation de l'Interface Runnable

```
class Worker2 implements Runnable
{
   public void run() {
      System.out.println("I Am a Worker Thread ");
   }
}
public class Second
{
   public static void main(String args[]) {
      Runnable runner = new Worker2();
      Thread thrd = new Thread(runner);
      thrd.start();

      System.out.println("I Am The Main Thread");
   }
}
```

Etats des Thread Java



Lier des Threads

```
class JoinableWorker implements Runnable
{
   public void run() {
      System.out.println("Worker working");
   }
}

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("Worker done");
   }
}
```

Suppression d'un Thread

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

. . .

// now interrupt it
Thrd.interrupt();
```

Suppression d'un Thread

```
public class InterruptibleThread implements Runnable
{
   public void run() {
      while (true) {
         /**
          * do some work for awhile
          */

         if (Thread.currentThread().isInterrupted()) {
            System.out.println("I'm interrupted!");
            break;
         }
      }

      // clean up and terminate
   }
}
```

Données Spécifiques à un Thread

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

   public static void transaction() {
      try {
         /**
          * some operation where an error may occur
          */
      catch (Exception e) {
         errorCode.set(e);
      }
   }

   /**
    * get the error code for this transaction
    */
   public static Object getErrorCode() {
      return errorCode.get();
   }
}
```

Données Spécifiques à un Thread

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

   public void run() {
      provider.transaction();
      System.out.println(provider.getErrorCode());
   }
}
```

Problème du Producteur-Consommateur

```
public class Factory
{
   public Factory()   {
      // first create the message buffer
      Channel mailBox = new MessageQueue();

      // now create the producer and consumer threads
      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();
   }
}
```

Thread Producteur

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

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

   public void run() {
      Date message;

      while (true) {
         SleepUtilities.nap();
         message = new Date();
         System.out.println("Producer produced " + message);

         // produce an item & enter it into the buffer
         mbox.send(message);
      }
   }
}
```

Thread Consommateur

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

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

   public void run() {
      Date message;

      while (true) {
         SleepUtilities.nap();
         // consume an item from the buffer
         System.out.println("Consumer wants to consume.");

         message = (Date)mbox.receive();
         if (message != null)
            System.out.println("Consumer consumed " + message);
      }
   }
}
```