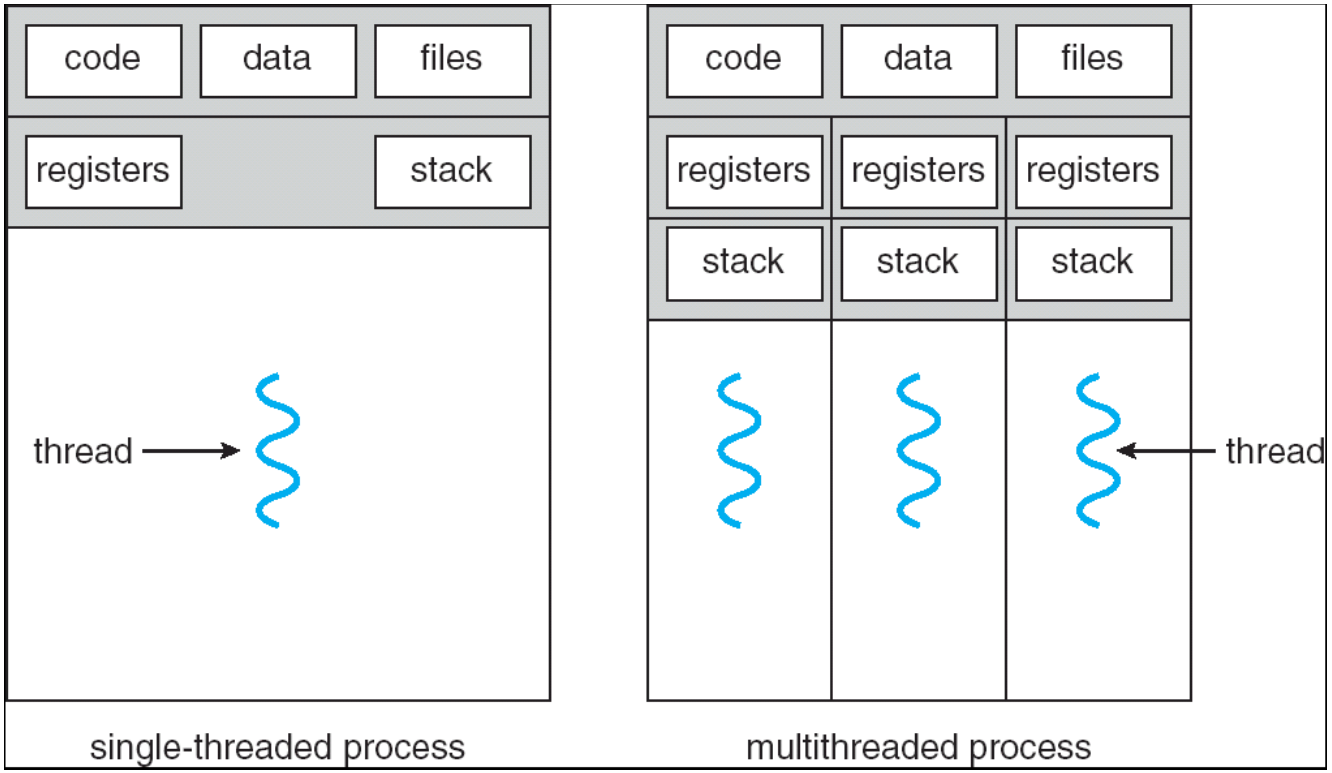
Capítulo 4: Threads

Capítulo 4: Threads

- Visão geral
- Modelos de multithreading
- Questões de threading
- Pthreads
- Threads do Windows XP
- Threads do Linux
- Threads Java

Processos de única e múltiplas threads



Benefícios

- Responsividade
- Compartilhamento de recursos
- Economia
- Utilização de arquiteturas multiprocessadas

Threads de usuário e kernel

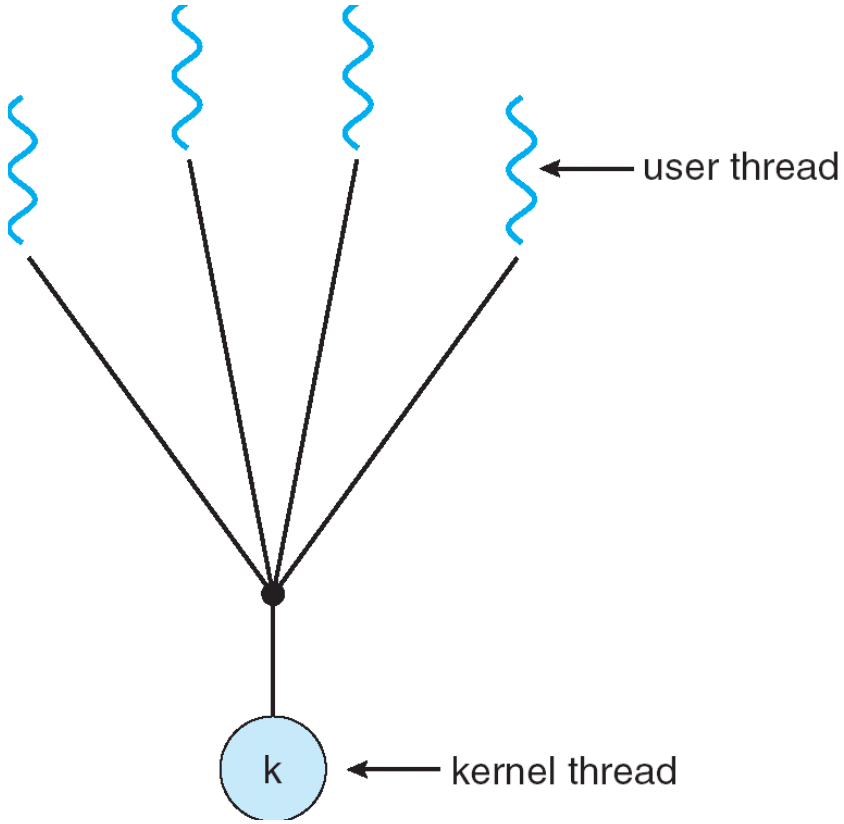
- Threads do usuário – Gerenciamento de thread feito pela biblioteca de threads em nível de usuário.
- Threads do kernel - Threads admitidas diretamente pelo kernel.

Modelos de multithreading

Mapeamento entre threads do usuário e threads do kernel:

- Muitos-para-um
- Um-para-um
- Muitos-para-muitos

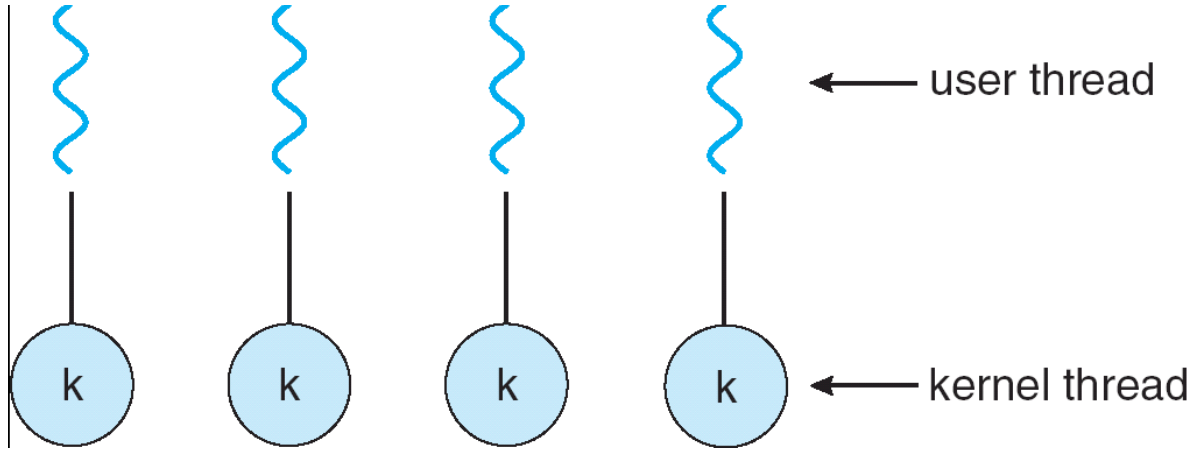
Modelo muitos-para-um



Muitos-para-um

- Muitas threads em nível de usuário mapeadas para única thread do kernel
- O processo inteiro é bloqueado se uma thread fizer uma chamada de sistema bloqueante.
- Só uma thread pode acessar o kernel por vez. Por isso, múltiplas threads não podem ser executadas em paralelo em processadores diferentes.
- Exemplos:
  - Solaris Green Threads
  - GNU Portable Threads

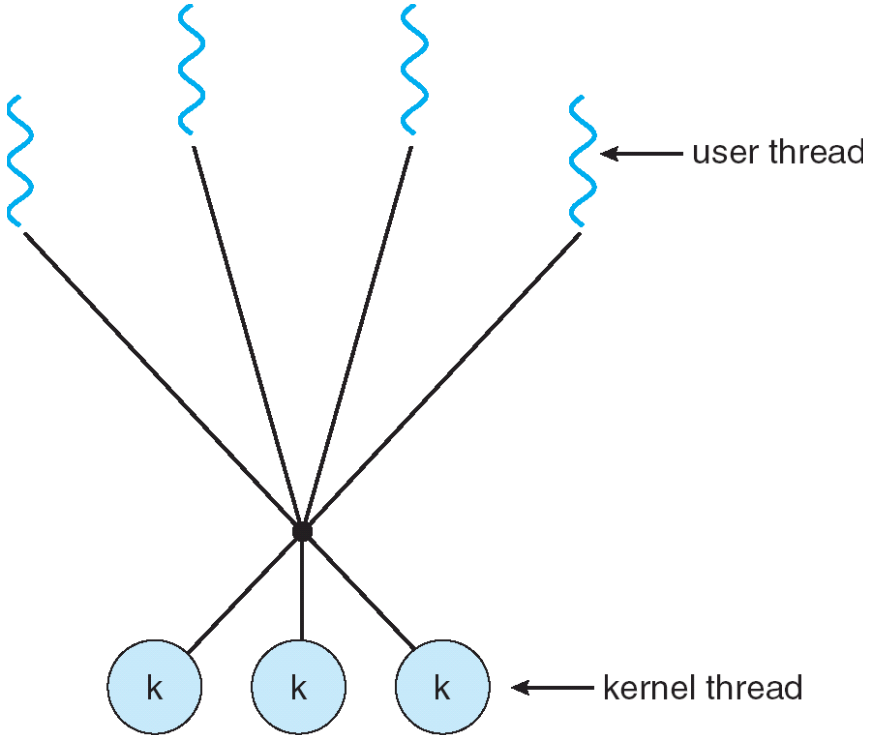
Modelo um-para-um



Um-para-um

- Cada thread em nível de usuário é mapeada para thread do kernel
- Maior concorrência do que o muitos-para-um (não bloqueia o processo todo quando uma thread faz uma chamada de sistema bloqueante)
- Permite rodar threads em paralelo em processadores diferentes
- Porém, custo maior, pois tem que criar uma thread de kernel para cada thread de usuário
- Exemplos: Windows NT/XP/2000, Linux, Solaris 9 em diante

Modelo muitos-para-muitos



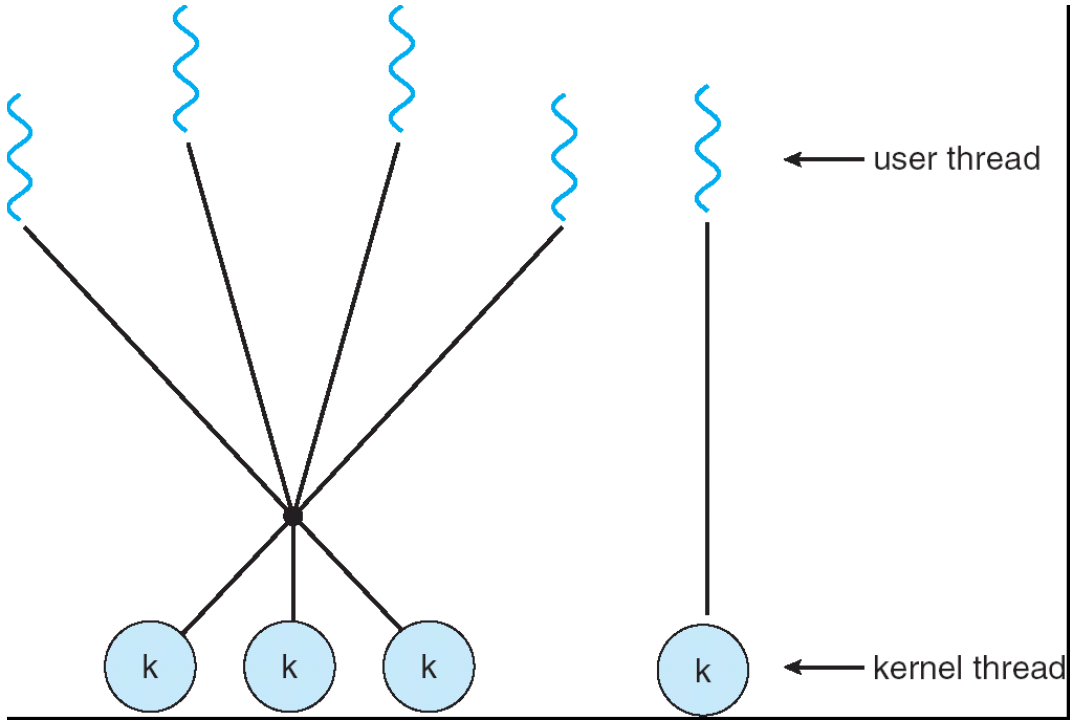
Modelo muitos-para-muitos

- Permite que muitas threads em nível de usuário sejam mapeadas para muitas threads do kernel
- Permite que o sistema operacional crie um número suficiente de threads do kernel
- Combina “o melhor” dos dois modelos anteriores (maior concorrência, multiprocessado e com menor custo)
- Solaris antes da versão 9
- Windows NT/2000 com o pacote *ThreadFiber*

Modelo de nível dois

- “Junção” dos modelos muitos-para-muitos e um-para-um.
- Exemplos
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 e mais antigos

Modelo de nível dois



Threads Java

- Threads Java são gerenciadas pela JVM
- Threads Java podem ser criadas Implementando a interface Runnable

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

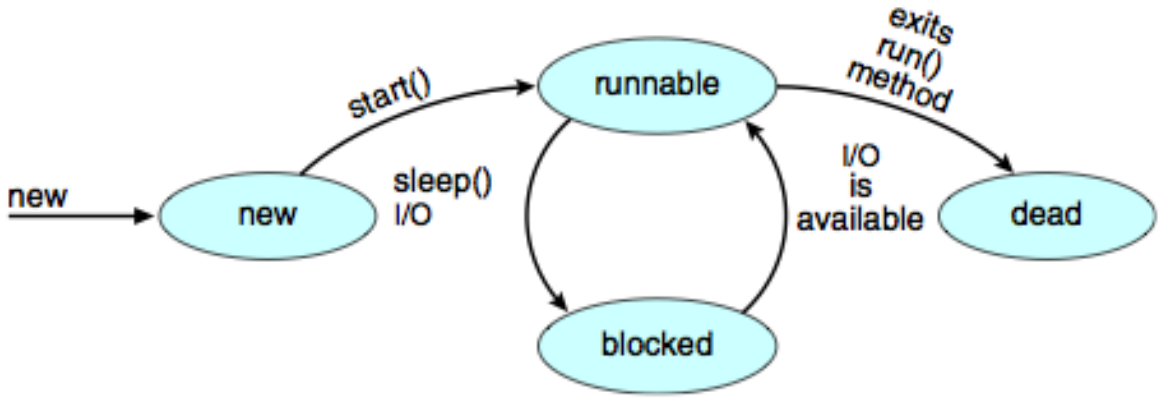
Threads Java - Programa de exemplo

```
class MutableInteger
{
  private int value;
  public int getValue() {
    return value;
  }
  public void setValue(int value) {
    this.value = value;
  }
}

class Summation implements Runnable
{
  private int upper;
  private MutableInteger sumValue;
  public Summation(int upper, MutableInteger sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }
  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
       sum += i;
    sumValue.setValue(sum);
  }
}
```

Threads Java - Programa de exemplo

```
public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else {
        // create the object to be shared
        MutableInteger sum = new MutableInteger();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sum));
        thrd.start();
        try {
          thrd.join();
          System.out.println
                  ("The sum of "+upper+" is "+sum.getValue());
        } catch (InterruptedException ie) { }
      }
    }
    else
      System.err.println("Usage: Summation <integer value>");
    }
}
```

Estados de threads Java



Threads Java – Produtor-Consumidor

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

       // Create the producer and consumer threads and pass
       // each thread a reference to the mailBox object.
       Thread producerThread = new Thread(
         new Producer(mailBox));
       Thread consumerThread = new Thread(
         new Consumer(mailBox));

       // Start the threads.
       producerThread.start();
       consumerThread.start();
    }

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

Threads Java - Produtor-Consumidor

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

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

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // produce an item and enter it into the buffer
         message = new Date();

         System.out.println("Producer produced " + message);
         mbox.send(message);
      }
   }
}
```

Threads Java - Produtor-Consumidor

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

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

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();

         // consume an item from the buffer
         message = (Date)mbox.receive();

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

Questões de threading

- Semântica das chamadas do sistema **fork()** e **exec()**
- Cancelamento de thread
- Tratamento de sinal
- Pools de thread
- Dados específicos da thread
- Ativações do escalonador

Cancelamento de thread

- Terminando uma thread antes que ele tenha terminado
- Duas técnicas gerais:
  - **Cancelamento assíncrono** termina a thread de destino imediatamente
  - **Cancelamento adiado** permite que a thread de destino verifique periodicamente se ela deve ser cancelada

Cancelamento de thread

Cancelamento adiado em Java
Interrompendo uma thread

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

Cancelamento de thread

Cancelamento adiado em Java
Verificando status da interrupção

```
class InterruptibleThread implements Runnable
{
   /**
    * This thread will continue to run as long
    * as it is not interrupted.
    */
   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
   }
}
```

Tratamento de sinal

- Sinais são usados em sistemas UNIX para notificar um processo de que ocorreu um evento em particular

- Um **manipulador de sinal** é usado para processar sinais
  1. Sinal é gerado por evento em particular
  2. Sinal é entregue a um processo
  3. Sinal é tratado
- Opções:
  - Entregar o sinal à thread para a qual o sinal se aplica
  - Entregar o sinal a cada thread no processo
  - Entregar o sinal a certas threads no processo
  - Atribuir uma área específica para receber todos os sinais para o processo

Pools de threads

- Criam uma série de threads em um pool onde esperam trabalho
- Vantagens:
  - Em geral, ligeiramente mais rápido para atender uma solicitação com uma thread existente do que criar uma nova thread
  - Coloca um limite (tamanho do pool) para a quantidade de threads alocadas a um processo, evitando excesso de threads ativas.

Pools de threads

- Java oferece 3 arquiteturas de pool de threads:

1. **Executor de única thread** - pool de tamanho 1
   - `static ExecutorService newSingleThreadExecutor()`
2. **Executor de thread fixo** - pool de tamanho fixo.
   - `static ExecutorService newFixedThreadPool(int nThreads)`
3. **Pool** de threads em cache - pool de tamanho ilimitado
   - `static ExecutorService newCachedThreadPool()`

Pools de threads

Uma tarefa a ser atendida em um pool de threads

```
public class Task implements Runnable
{
   public void run() {
    System.out.println("I am working on a task.");
    . . .
   }
}
```

Pools de threads

Criando um pool de threads em Java

```
import java.util.concurrent.*;

public class TPExample
{
   public static void main(String[] args) {
      int numTasks = Integer.parseInt(args[0].trim());

      // create the thread pool
      ExecutorService pool = Executors.newCachedThreadPool();

      // run each task using a thread in the pool
      for (int i = 0; i < numTasks; i++)
         pool.execute(new Task());

      // Shut down the pool. This shuts down the pool only
      // after all threads have completed.
      pool.shutdown();
   }
}
```

Dados específicos da thread

- Permite que cada thread tenha sua própria cópia dos dados (ex: cada thread vai tratar uma transação e, portanto, guarda o número da respectiva transação)
- Útil quando você não tem controle sobre o processo de criação de thread (ex: ao usar um pool de threads, uma mesma thread pode ser reutilizada para realizar o trabalho de 2 ou mais “threads”).

Dados específicos do thread

Dados específicos do thread em Java

```
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();
   }
}
```

Dados específicos de thread

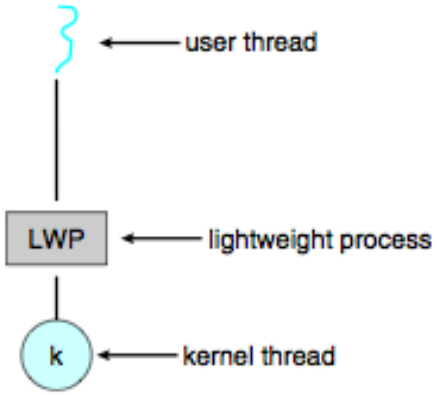
```
class Worker implements Runnable {
   private static Service provider;

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

Exemplo:

- Threads T1 e T2 invocam `transaction()`.
- T1 lança a exceção E1 e T2 lança a exceção E2.
- Os valores de `errorCode` para T1 e T2 serão E1 e E2, respectivamente.

Ativação do escalonador

- Os modelos M:M e de dois níveis exigem comunicação com o kernel para manter o número apropriado de threads de kernel alocados ao processo
- A **ativação do escalonador** [de threads do usuário, que é implementado pela biblioteca de threads] ocorre através de **upcalls** – um mecanismo de comunicação do kernel para a biblioteca de threads (ex: quando há um deadlock entre as threads de usuário, o kernel pode fazer um upcall)
- Para a thread de usuário, o LWP (a estrutura de dados que está entre a thread de usuário e a thread do kernel) representa um "processador virtual".
- O LWP roda no espaço do processo (não precisa mudar para modo kernel) – melhora o desempenho.



Pthreads

- Uma API padrão POSIX (IEEE 1003.1c) para criação e sincronismo de thread
- A API especifica o comportamento da biblioteca de threads, a implementação fica para o desenvolvimento da biblioteca
- Comum em sistemas operacionais UNIX (Solaris, Linux, Mac OS X)

Final do Capítulo 4