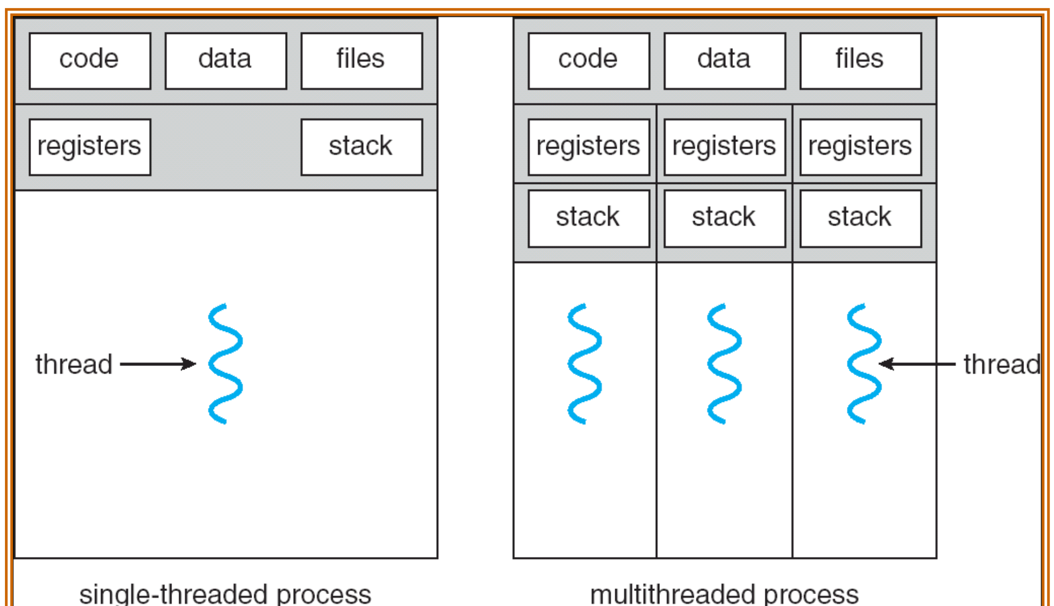
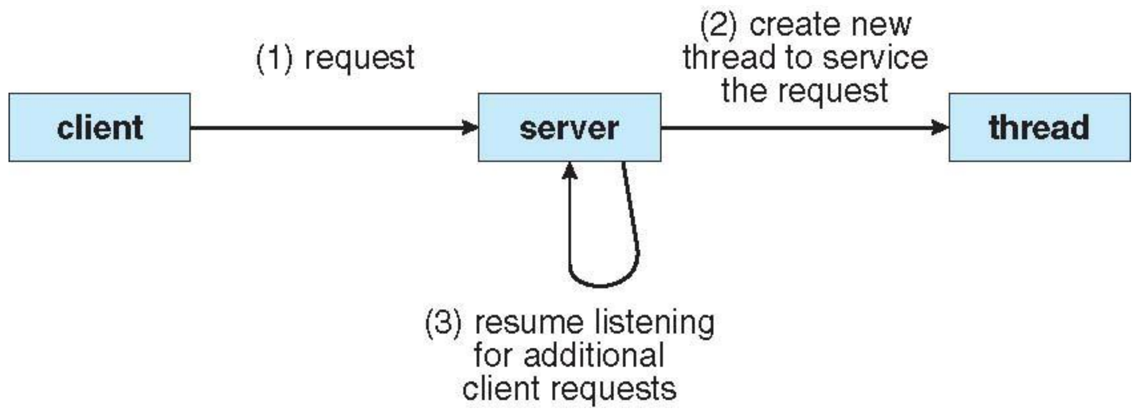
Threads

Overview (1)

- Thread: basic unit of CPU utilisation
  - Comprises of thread ID, program counter, register set and a stack
  - Threads within a process share: code and data section, OS resources (open files and signals)
  - Multithreaded applications & OS

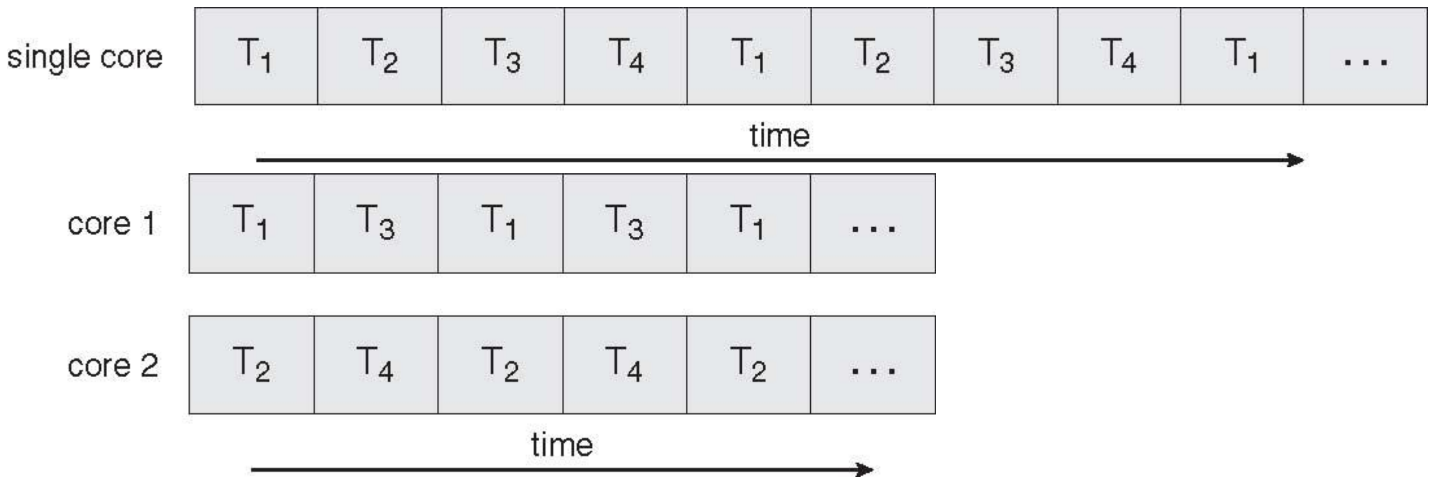


Overview (2)



- Benefits
  - Responsiveness
  - Resource sharing
  - Economy (creation and context switch, up to 30 time difference)
  - Utilisation of multiprocessor architectures

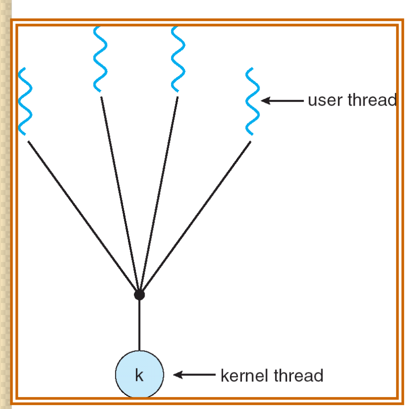
Overview (3)

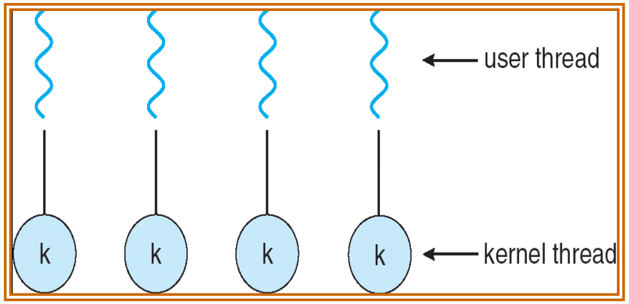


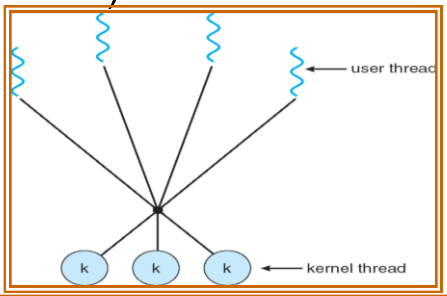
- Multi-core programming challenges
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

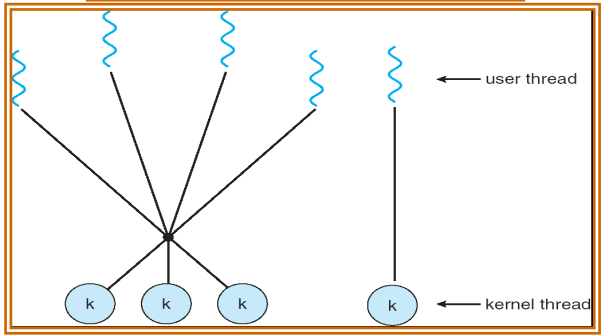
Multithreading Models

- User versus kernel threads
  - Many-to-one: thread management in user space – efficient, whole process blocks in blocking system calls, unable to exploit multiprocessors
    - Solaris Green threads & GNU Portable threads
  - One-to-one: allows more concurrency, cost of creating both threads and limit on threads
    - Linux & Windows
  - Many-to-many: multiplexing of user threads, application or machine specific allocation, any number of user threads with more concurrency
    - Windows with ThreadFiber package
    - Two-level model – IRIX, HP-UX, Tru64 Unix, Solaris (before 9)









Thread Libraries

- Thread library: an API for creating and managing threads
  - User space versus kernel space
  - Local function call versus kernel system call
  - POSIX Pthreds (either), Win32 (kernel), Java (implemented using the thread library of the host system)

Java Threads (1)

- All programs have at least one thread
- Thread creation
  - Extend the Thread class
  - Implementing the Runnable interface

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

  - Creating a thread object does not create a thread, calling the start() method does
  - Data sharing through passing of references to the shared object
  - Daemon and non daemon threads – setDaemon()

Java Threads (2)

```
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);
  }
}
```

```
class Sum {
	private int sum;

	public int getSum(){
		return sum;
	}

	public void setSum(int sum) {
		this.sum = sum;
	}
}
```

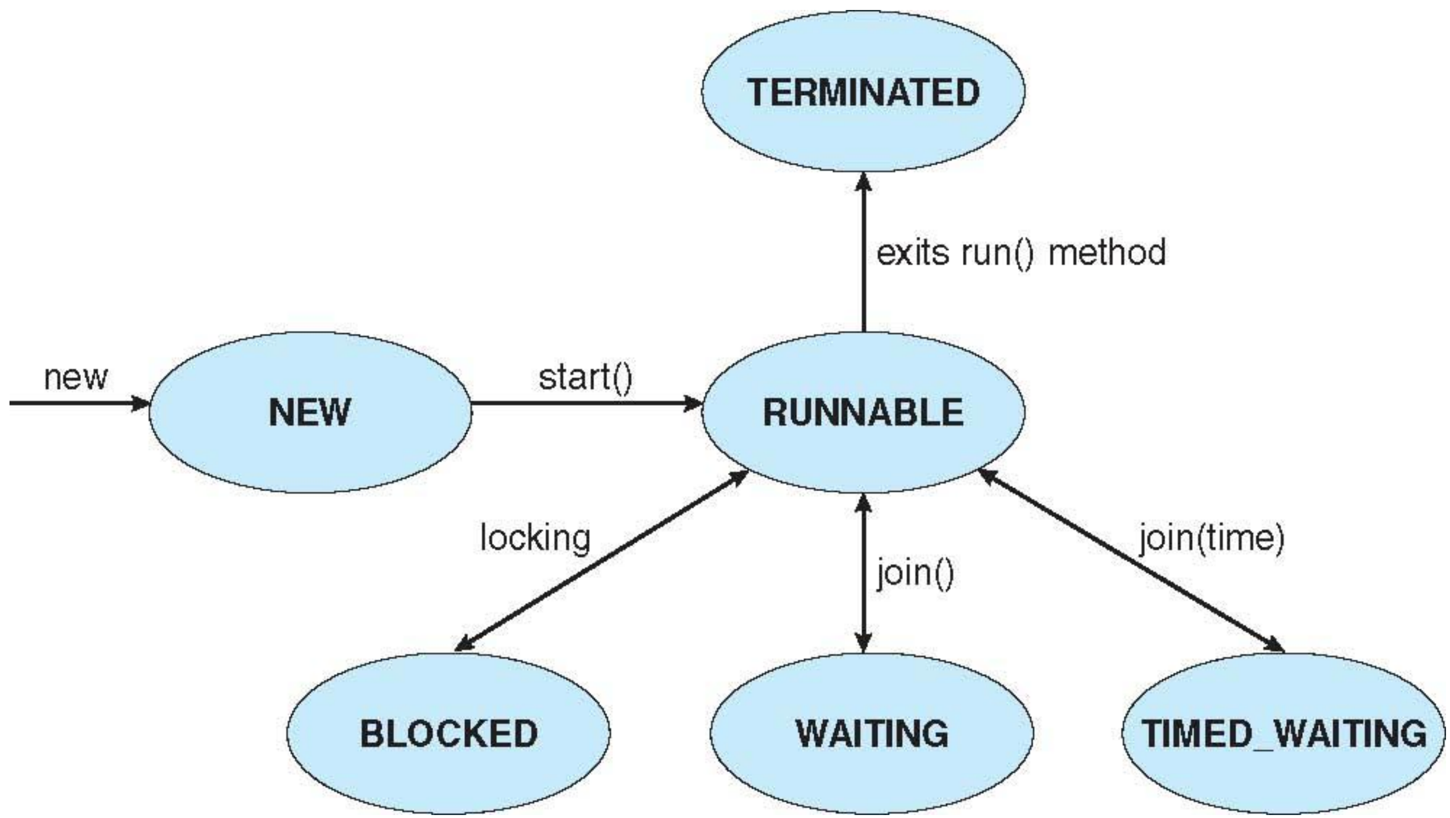
Java Threads (3)

```
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>");
    }
}
```

Java Threads (4)

- JVM and host OS
  - The specification does not indicate how threads are to be mapped
    - Windows XP (one-to-one model) maps each Java thread to a kernel thread
    - Solaris initially used the many-to-one model (Green threads), later the one-to-one model
  - Java thread library relates to host OS thread library

Java Threads (5)



`isAlive()`

Java Threads (6)

```
public interface Channel
{
   // Send a message to the channel
   public abstract void send(Object item);

   // Receive a message from the channel
   public abstract Object receive();
}
```

```
public class MessageQueue implements Channel
{
   private Vector queue;

   public MessageQueue() {
      queue = new Vector();
   }

   // This implements a nonblocking send
   public void send(Object item) {
      queue.addElement(item);
   }

   // This implements a nonblocking receive
   public Object receive() {
      if (queue.size() == 0)
         return null;
      else
         return queue.remove(0);
   }
}
```

```
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();
   }
}
```

Java Threads (7)

```
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);
      }
   }
}
```

```
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);
      }
   }
}
```

Threading Issues (1)

- fork() and exec()
  - Are the all threads or just the calling thread duplicated? – either
  - exec() replaces the entire process including all threads
  - If exec() is to be called then only replicate the calling thread

Threading Issues (2)

- Cancellation: terminating a thread before it has finished
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately
    - stop() method, but deprecated
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled

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

Threading Issues (3)

```
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
   }
}
```

Threading Issues (4)

- Signal handling
  - UNIX - notify process that an event occurred
    - Synchronous (e.g. illegal memory access, division by 0) – delivered to the offending process
    - Asynchronous (e.g. <control><C>, timer expiration) – sent to another process
  - Every signal has a default signal handler (run by the kernel) that may be overridden by a user-defined signal handler
  - Multithreading option:
    - Deliver to the thread the signal applies
    - Deliver to every thread of the process
    - Deliver to certain threads of the process
    - Assign a thread to receive all signals for the process
    - Blocking signals
    - Only one thread receives
  - Windows – signal emulation with asynchronous procedure calls (APCs) (associated to particular threads)

Threading Issues (5)

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

Threading Issues (6)

- Java provides 3 thread pool architectures:

1. Single thread executor - pool of size 1.
   - `static ExecutorService newSingleThreadExecutor()`
2. Fixed thread executor - pool of fixed size.
   - `static ExecutorService newFixedThreadPool(int nThreads)`
3. Cached thread pool - pool of unbounded size
   - `static ExecutorService newCachedThreadPool()`

Threading Issues (7)

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

```
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();
  }
}
```

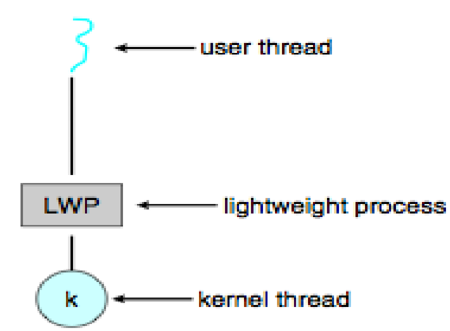
Threading Issues (8)

- Thread specific data
- Scheduler activations
  - Many-to-many and two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
  - Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
  - This communication allows an application to maintain the correct number kernel threads

```
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();
   }
}
```



Linux threads

- Does not distinguish between process and threads – tasks
  - Unique kernel data structure for each task that instead of storing data contains pointers to other data structures
- Thread creation: clone()
  - No flags – similar to fork()

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

For contemplation (1)

- Describe the actions taken by a thread library to context switch between user-level threads.
- Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?
- Which of the following components of program state are shared across threads in a multithreaded process?
  - Register values
  - Heap memory
  - Global variables
  - Stack memory
- What resources are used when a thread is created? How do they differ from those used when a process is created?

For contemplation (2)

- The Java API provides several different thread-pool architectures:
  - newFixedThreadPool(int)
  - newCachedThreadPool()
  - newSingleThreadExecutor()
  - Discuss the merits of each.
- Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the clone() system call. However, many operating systems— such as Windows XP and Solaris—treat processes and threads differently. Typically, such systems use a notation wherein the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modelling processes and threads within the kernel.

For contemplation (3)

- Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processors in the system. Discuss the performance implications of the following scenarios.
  - The number of kernel threads allocated to the program is less than the number of processors.
  - The number of kernel threads allocated to the program is equal to the number of processors.
  - The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user level threads.