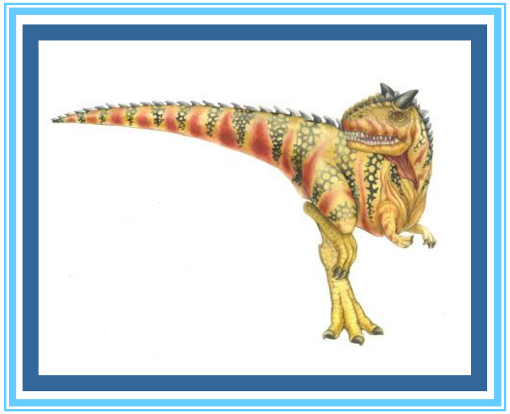
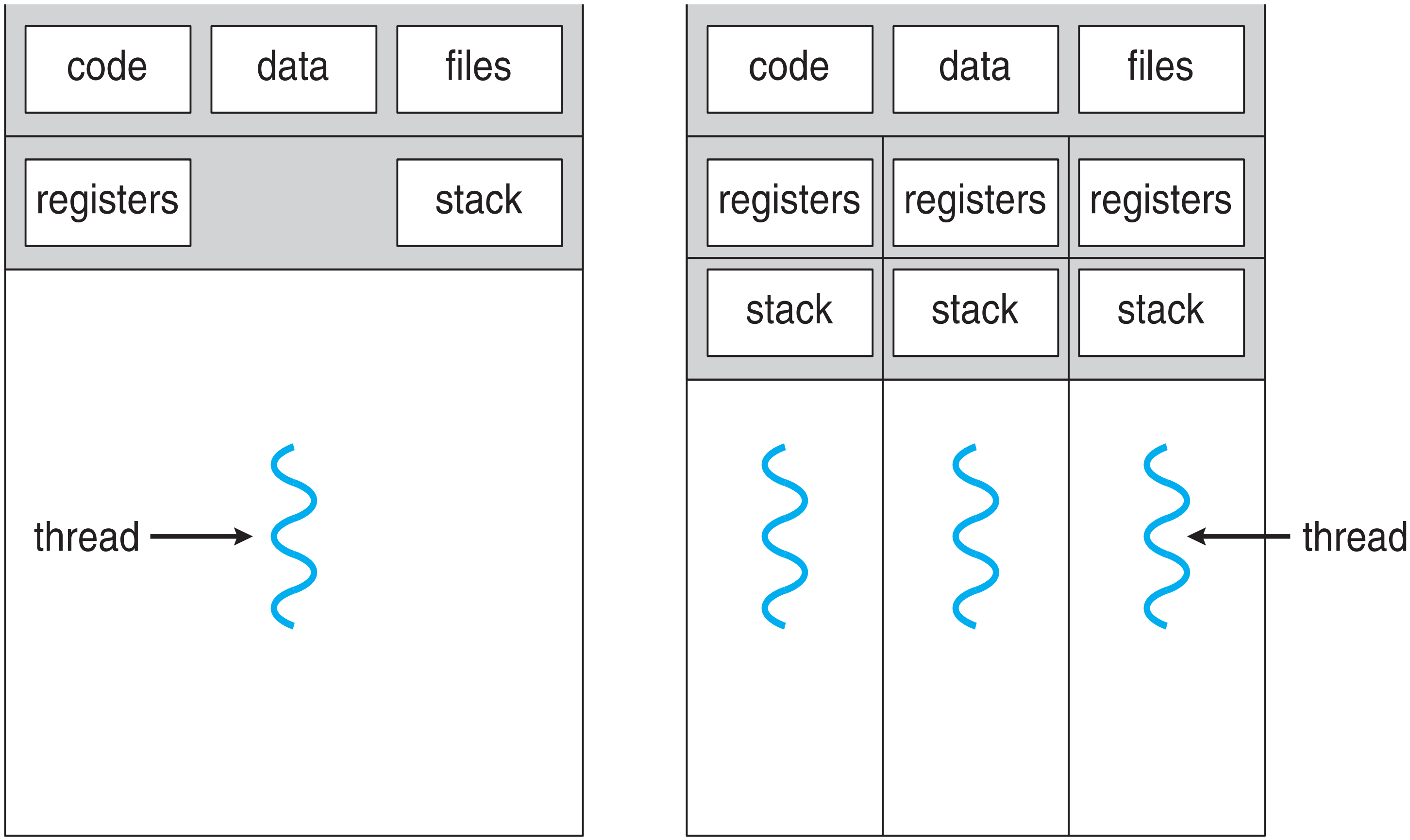
Java Threads (cont'd)



Agenda

- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading

Single and Multithreaded Processes



single-threaded process

multithreaded process

Benefits

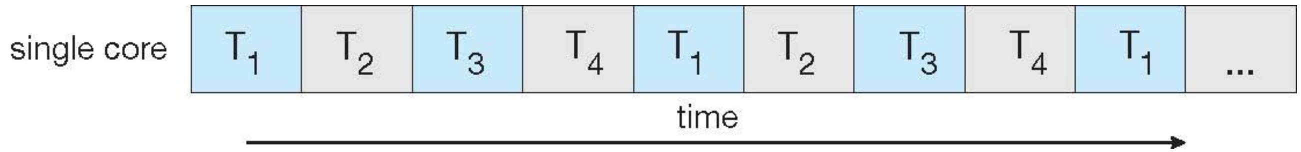
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

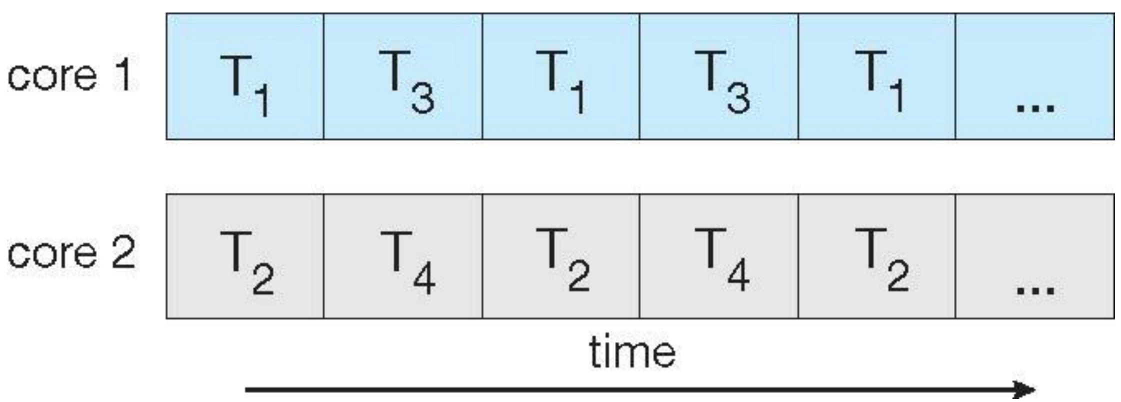
- **Multi-CPU systems**. Multiple CPUs are placed in the computer to provide more computing performance.
- **Multicore systems**. Multiple computing cores are placed on a single processing chip where each core appears as a separate CPU to the operating system
- Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.
- Consider an application with four threads.
  - On a system with a single computing core, concurrency means that the execution of the threads will be interleaved over time.
  - On a system with multiple cores, however, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



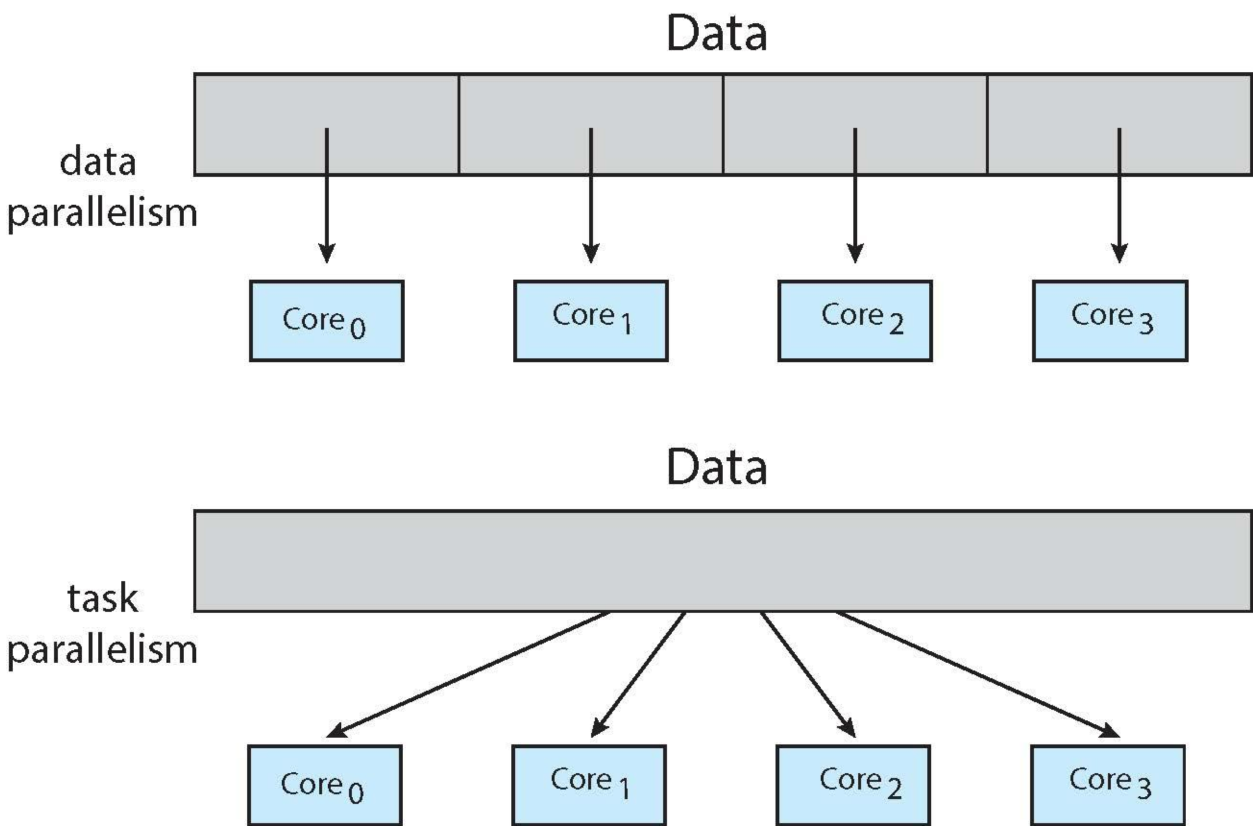
Multicore Programming (Cont.)

- There is a fine but clear distinction between concurrency and parallelism..
- A concurrent system supports more than one task by allowing all the tasks to make progress.
- In contrast, a system is parallel if it can perform more than one task simultaneously.
- Thus, it is possible to have concurrency without parallelism

Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As number of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Data and Task Parallelism

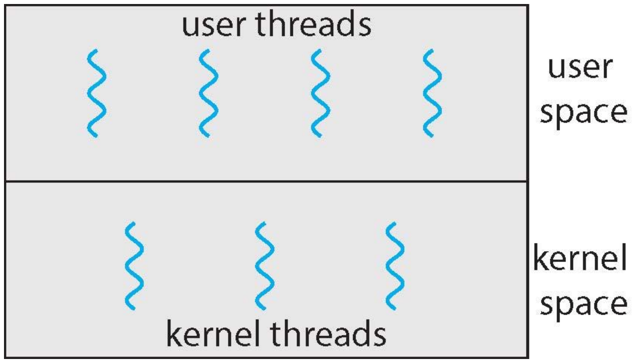


Multicore Programming

- **Multicore** or **multiprocessor** systems are placing pressure on programmers. Challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

User and Kernel Threads

- Support for threads may be provided at two different levels:
  - **User threads** - are supported above the kernel and are managed without kernel support, primarily by user-level threads library.
  - **Kernel threads** - are supported by and managed directly by the operating system.



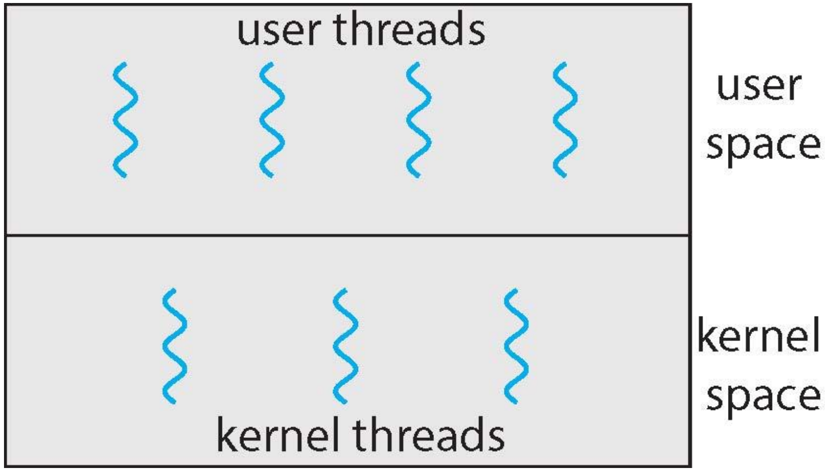
- Virtually all contemporary systems support kernel threads:
  - Windows, Linux, and Mac OS X

Relationship between user and Kernel threads

- Three common ways of establishing relationship between user and kernel threads:
  - Many-to-One
  - One-to-One
  - Many-to-Many

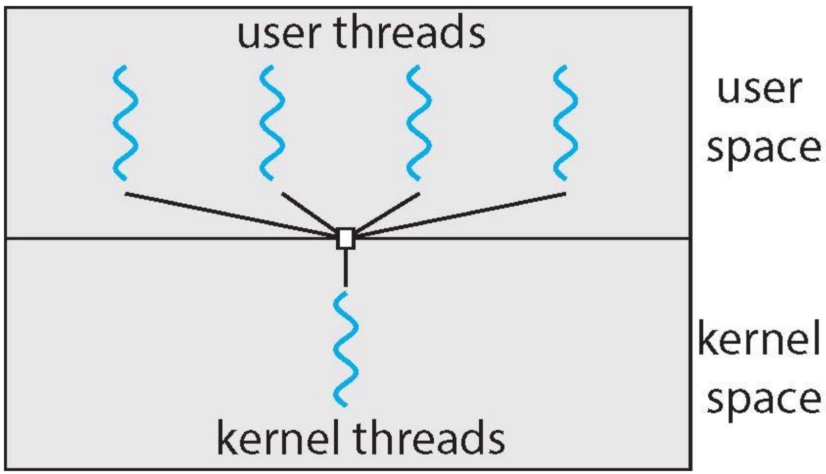
One-to-One Model

- Each user-level thread maps to a single kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



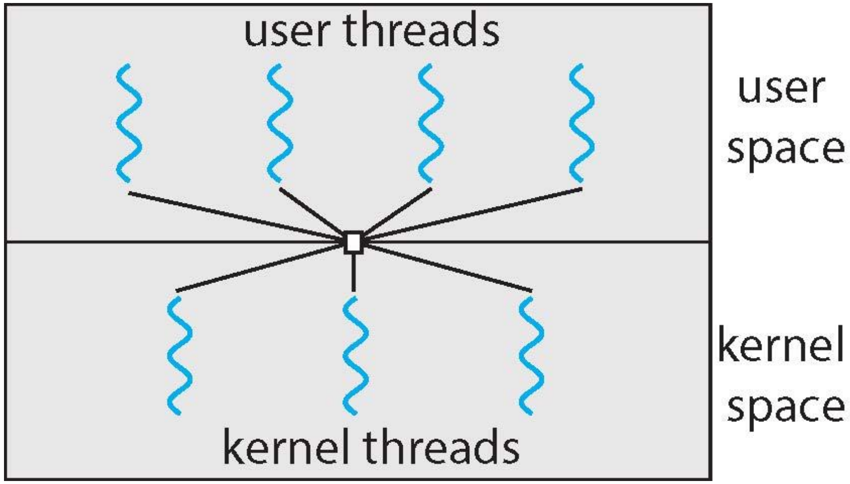
Many-to-One Model

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Two techniques for creating threads in a Java program.
  - One approach is to create a new class that is derived from the Thread class and to override its run() method.
  - An alternative   is to define a class that implements the Runnable} interface, as shown below

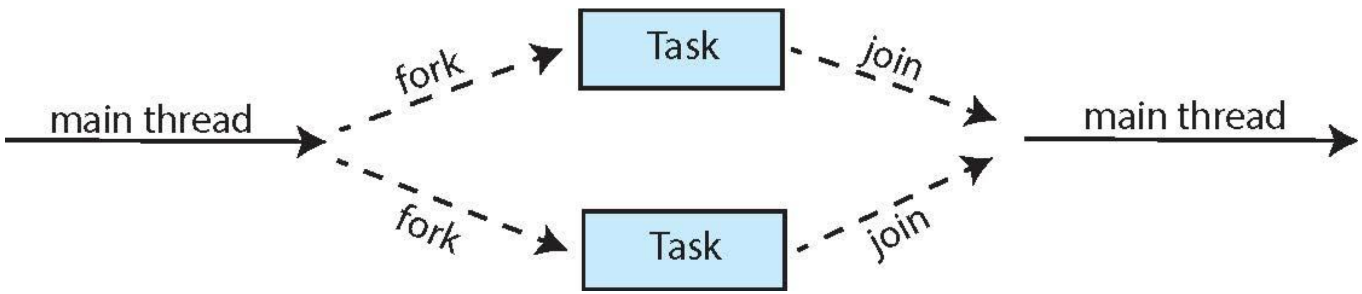
```
public interface Runnable
{
    public abstract void run();
}
```

- When a class implements Runnable, it must define a run() method. The code implementing the run(} method is what runs as a separate thread.

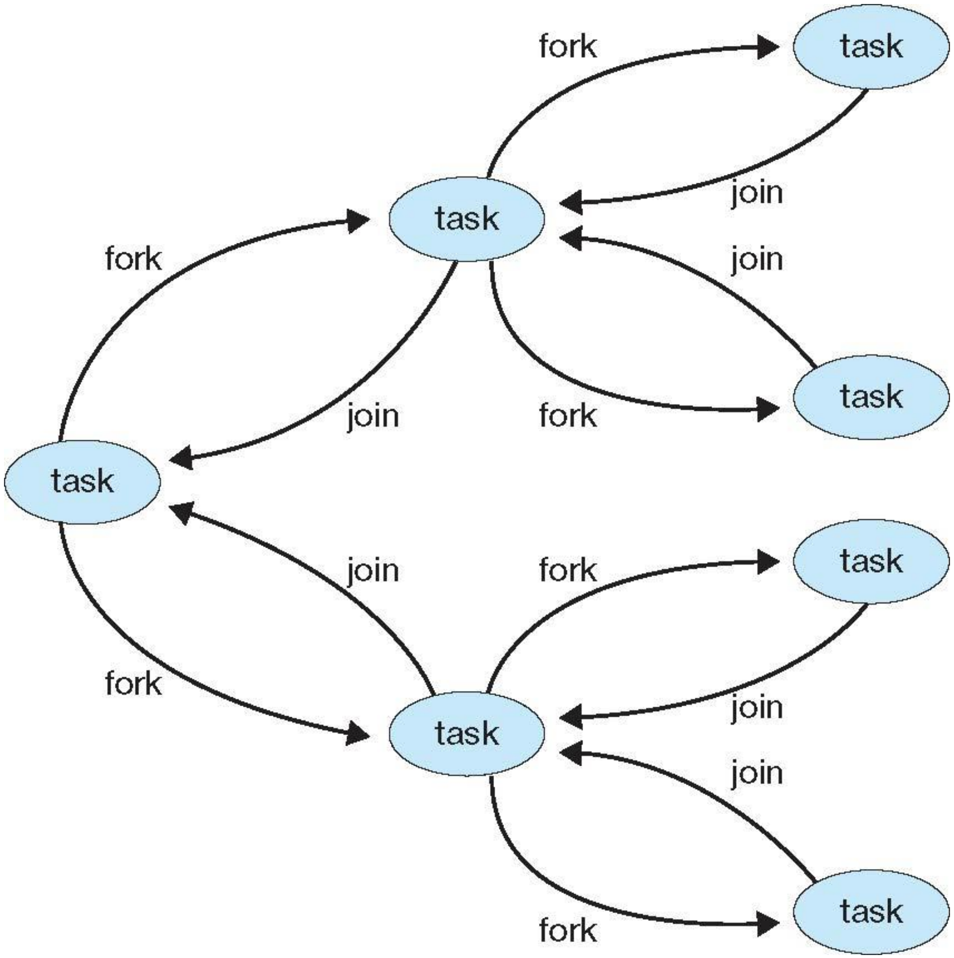
Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Methods explored
  - Thread Pools
  - Fork Join
  - OpenMP
  - Intel Thread Building Blocks
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **`java.util.concurrent`** package

Fork-Join Parallelism



Fork-Join in JAVA



Fork-Join Calculation using the Java API

```
import java.util.concurrent.*;
public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;
   private int begin;
   private int end;
   private int[] array;
   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }
```

Fork-Join Calculation using the Java API (Cont.)

```
protected Integer  compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];
         return sum;
      }
      else {
         int mid = (begin + end) / 2;
         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);
         leftTask.fork();
         rightTask.fork();
         return rightTask.join() + leftTask.join();
      }
   }
}
```