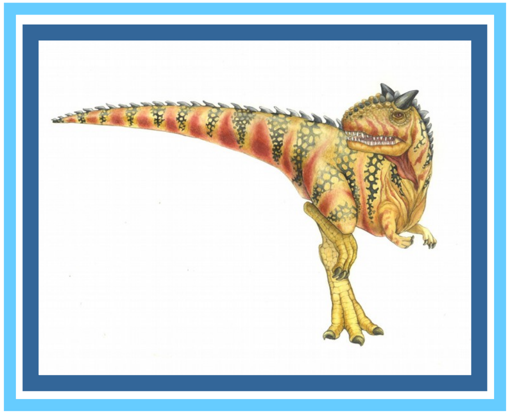
Chapter 4: Threads



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

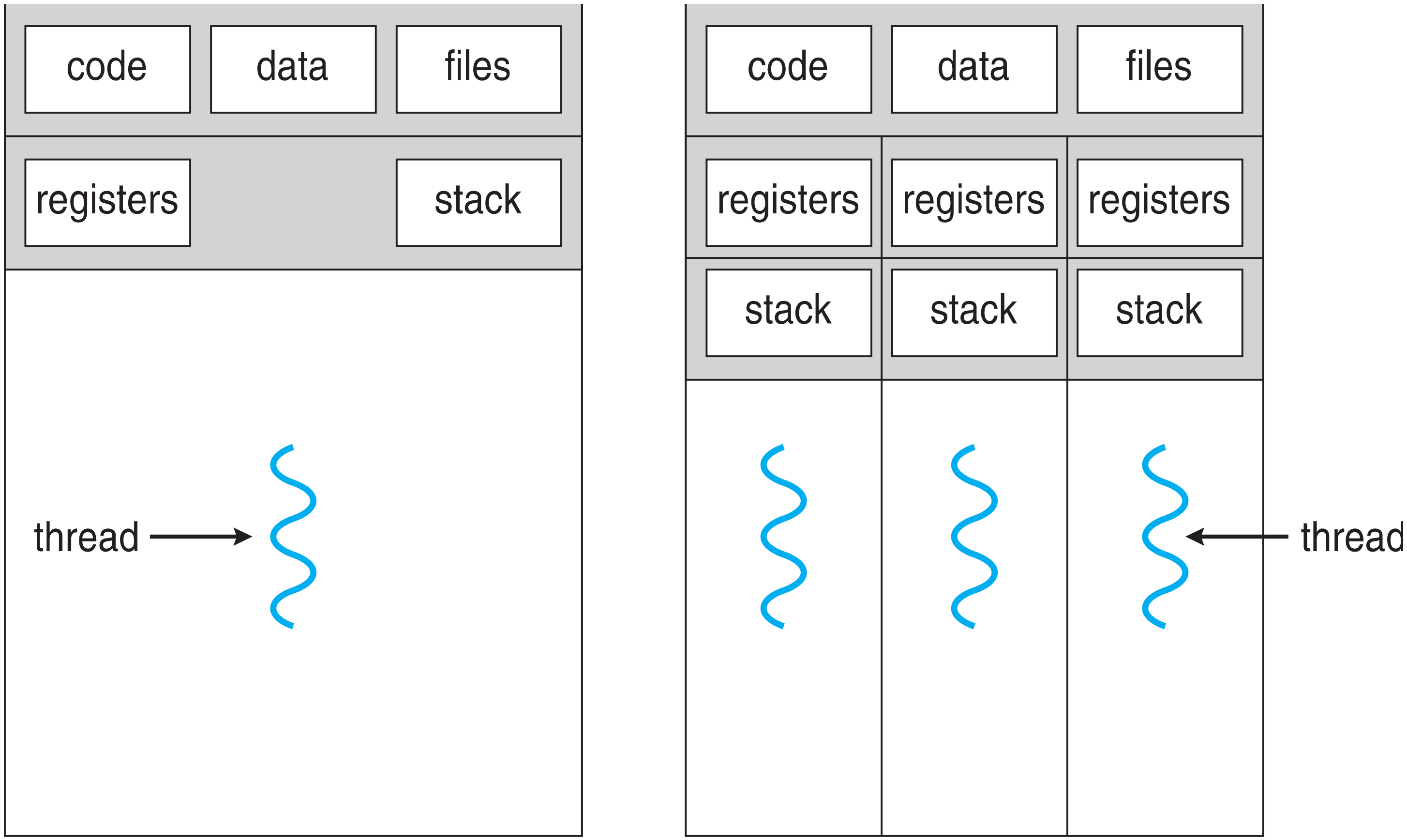
Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

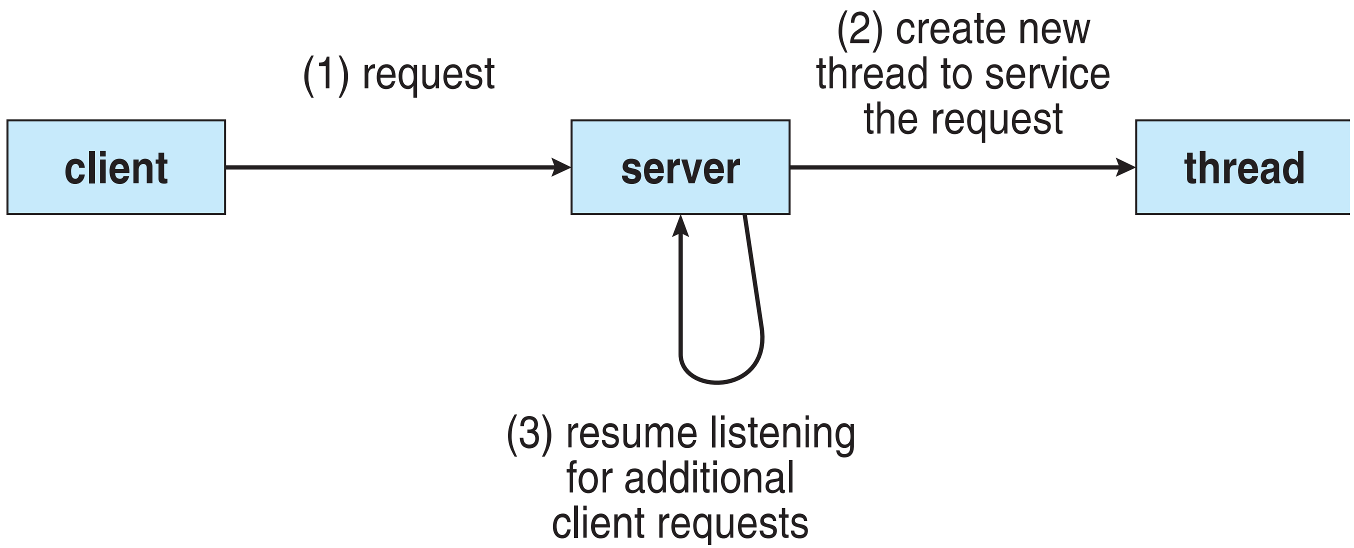
Single and Multithreaded Processes



single-threaded process

multithreaded process

Multithreaded Server Architecture



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

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.)

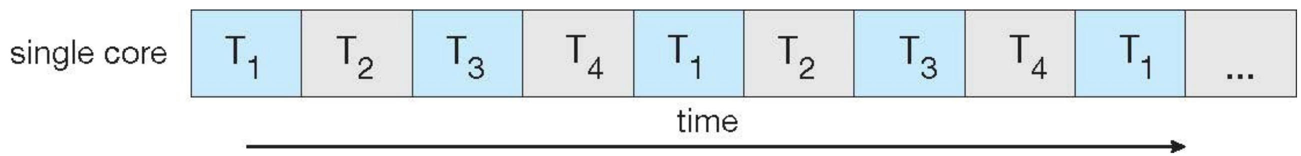
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As number of threads grows, so does architectural support for threading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

Multicore Programming

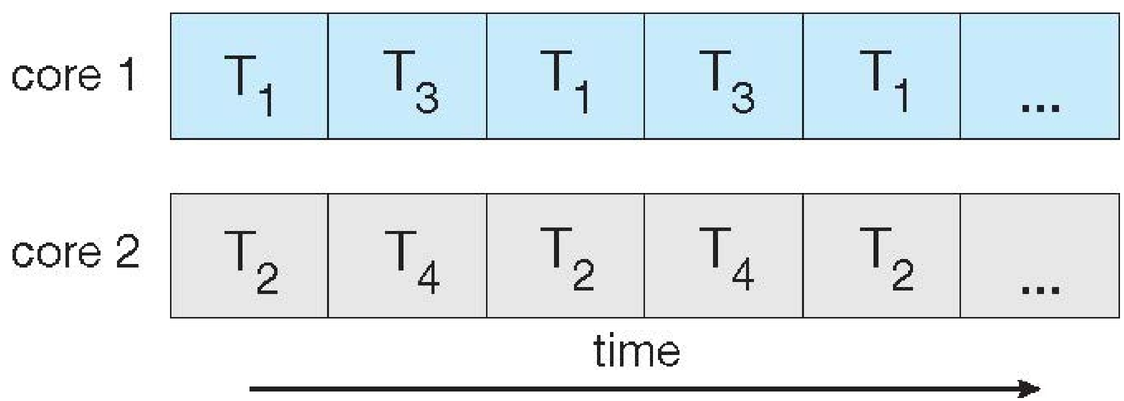
- **Multicore** or **multiprocessor** systems are placing pressure on programmers. Challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously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

Concurrency vs. Parallelism

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



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



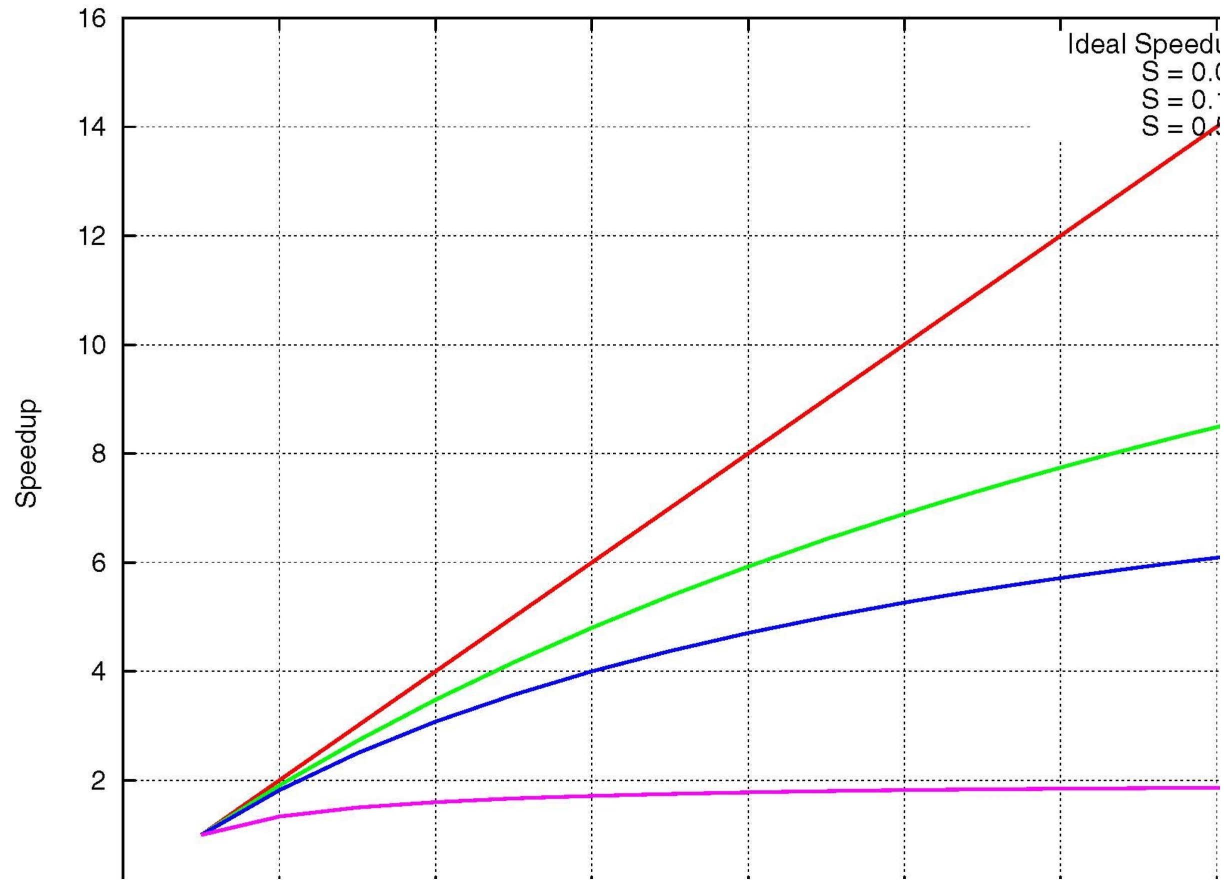
Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *N* processing cores and *S* is serial portion

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if an application is 75% parallel and 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*
- **Serial portion of an application has disproportionate effect on performance gained by adding additional cores**
- But does the law take into account contemporary multicore systems?

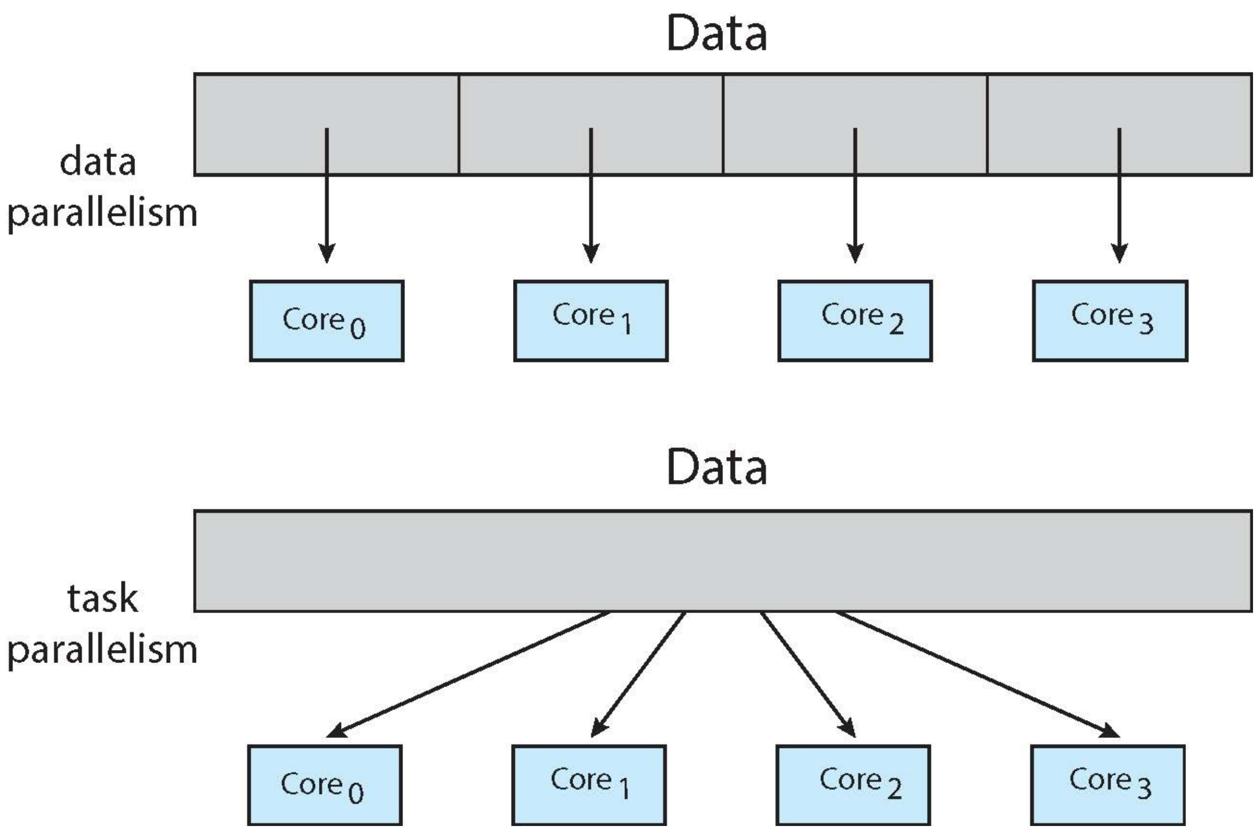
Figure Amdahl



Type of Parallelism

- **Data parallelism**. The focus is on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
  - Example -- summing the contents of an array of size N. On a single-core system, one thread would sum the elements 0 .. N-1. On a dual-core system, however, thread A, running on core 0, could sum the elements 0 .. N/2, while thread B, running on core 1, could sum the elements of N/2 .. N-1. The two threads would be running in parallel on separate computing cores.
- **Task parallelism.** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

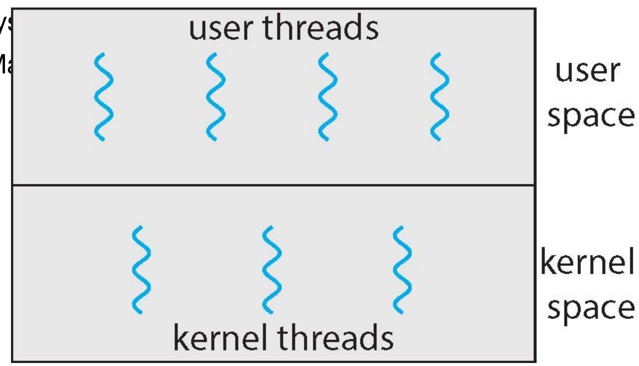
Data and Task Parallelism



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 sys
  - Windows, Linux, and Ma

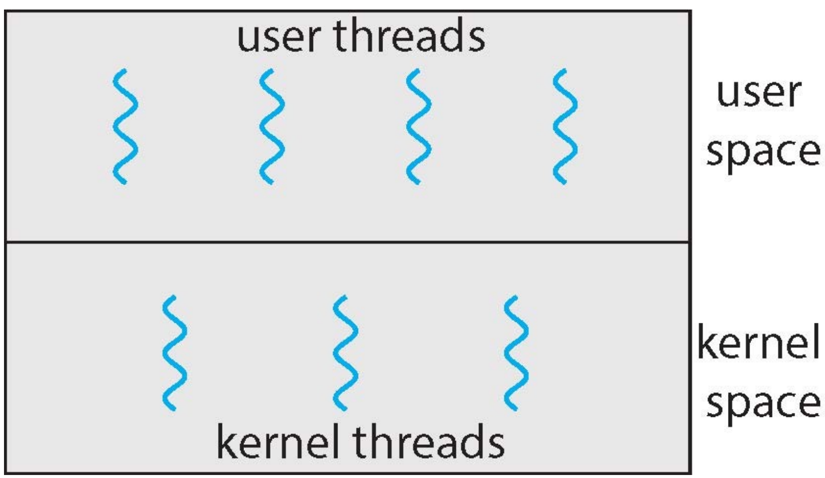


Relationship between user and Kernel threads

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

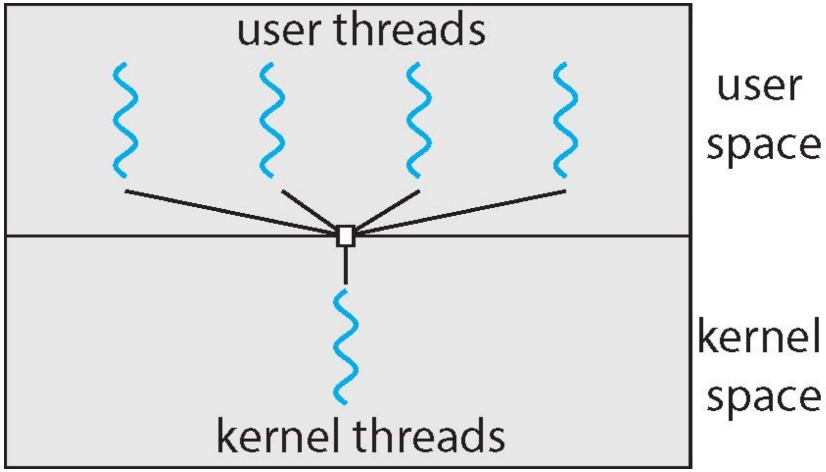
One-to-One Model

- 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



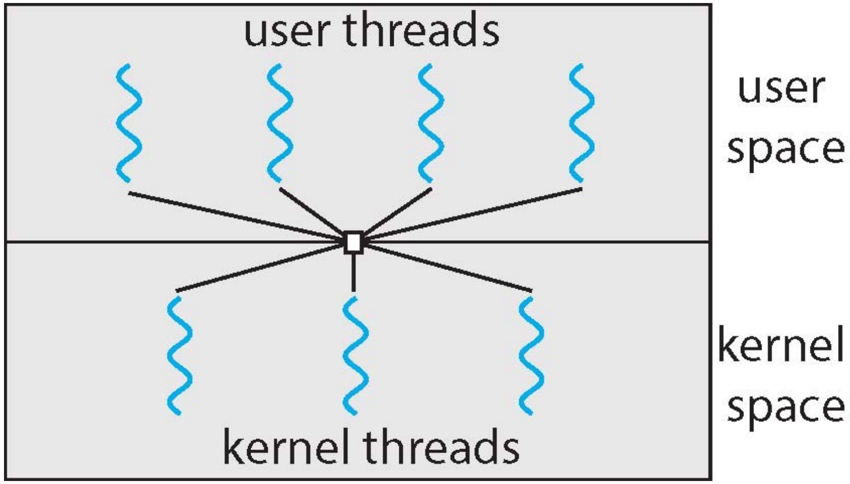
Many-to-One Model

- 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**



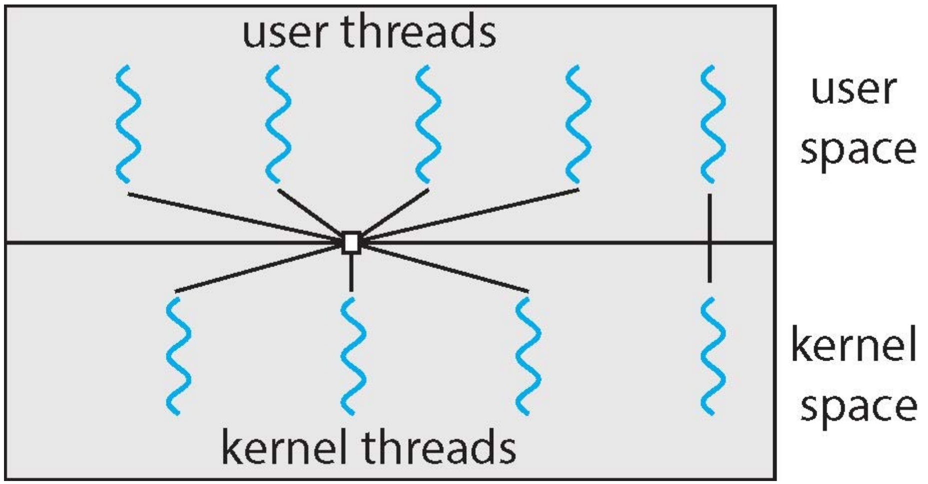
Many-to-Many Model

- 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



Two-level Model

- Similar to many-to-many, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



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

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

- Next two slides show a multithreaded C program that calculates the summation of a non-negative integer in a separate thread.
- In a Pthreads program, separate threads begin execution in a specified function. In the program, this is the runner() function.
- When this program starts, a single thread of control begins in main(). After some initialization, main() creates a second thread that begins control in the runner() function. Both threads share the global data sum.

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

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

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Pthreads Code for Joining Ten Threads

- The summation program in the previous slides creates a single thread.
- With multicore systems, writing programs containing several threads is common.
- Example a Pthreads program, for joining the threads:

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Windows Multithreaded C Program (Cont.)

```
/* create the thread */
ThreadHandle = CreateThread(
  NULL, /* default security attributes */
  0, /* default stack size */
  Summation, /* thread function */
  &Param, /* parameter to thread function */
  0, /* default creation flags */
  &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
   /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle,INFINITE);

  /* close the thread handle */
  CloseHandle(ThreadHandle);

  printf("sum = %d\n",Sum);
}
}
```

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) method. The code implementing the run(} method is what runs as a separate thread.

Java Multithreaded Program

- The Java program in the next slide shows a multithreaded program that determines the summation of a non-negative integer.
- The Summation class implements the Runnable interface.
- Thread creation is performed by creating an object instance of the Thread} class and passing the constructor a Runnable object.

Java Multithreaded Program (Cont.)

```java
class Sum
{
  private int sum;

  public int getSum() {
   return sum;
  }

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

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
   this.upper = upper;
   this.sumValue = sumValue;
  }

  public void run() {
   int sum = 0;
   for (int i = 0; i <= upper; i++)
      sum += i;
   sumValue.setSum(sum);
  }
}
```

Java Multithreaded Program (Cont.)

```
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 {
       Sum sumObject = new Sum();
       int upper = Integer.parseInt(args[0]);
       Thread thrd = new Thread(new Summation(upper, sumObject));
       thrd.start();
       try {
          thrd.join();
          System.out.println
                  ("The sum of "+upper+" is "+sumObject.getSum());
      } catch (InterruptedException ie) { }
      }
     }
     else
      System.err.println("Usage: Summation <integer value>"); }
}
```

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
- Three 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

Thread Pools

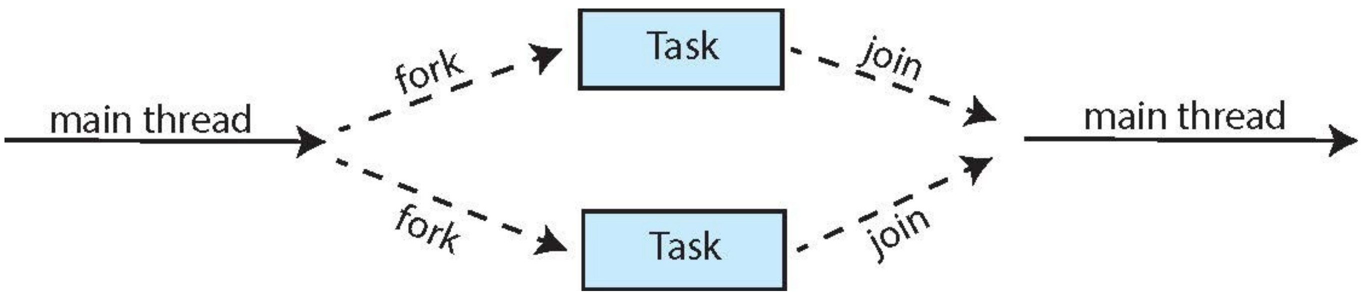
- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - That is, tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
   /*
    * this function runs as a separate thread.
    */
}
```

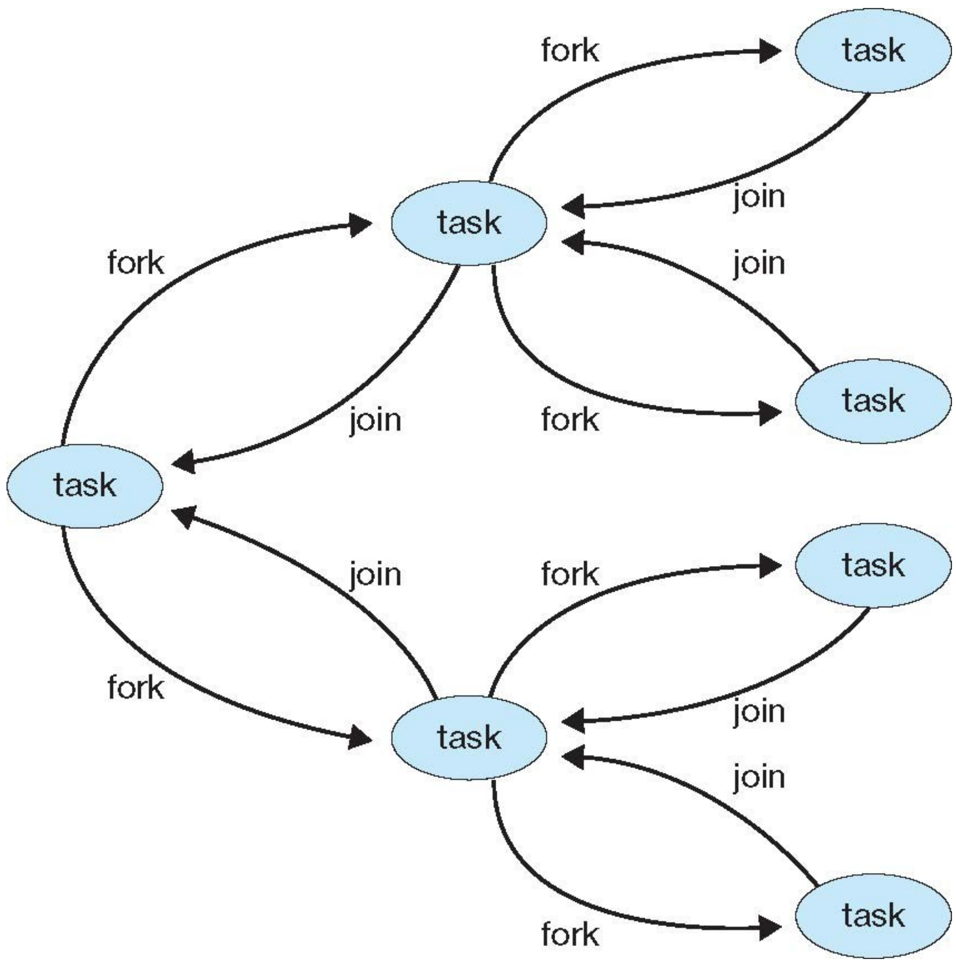
Creating a Thread Pool in Java

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

public class ThreadPoolExample
{
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 once all threads have completed */
      pool.shutdown();
}
```

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 - begin) / 2;
            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();
        }
    }
}
```

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

`#pragma omp parallel`

Create as many threads as there are cores

```
#pragma omp parallel for
   for(i=0;i<N;i++) {
     c[i] = a[i] + b[i];
}
```

Run for loop in parallel

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in "^{ }" - `^{ printf("I am a block"); }`
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch

- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called **main queue**
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIX systems have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads

Signal Handling

- **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 by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that the kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the threat 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 threat to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished
- The thread to be canceled is referred to as **target thread**
- Cancelation of a target thread may be handled using two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- The difficulty with cancellation occurs in situations where:
  - Resources have been allocated to a canceled thread.
  - A thread is canceled while in the midst of updating data it is sharing with other threads.

Thread Cancellation

- The operating system usually will reclaim system resources from a canceled thread but will not reclaim all resources.
- Canceling a thread asynchronously does not necessarily free a system-wide resource that is needed by others.
- Deferred cancellation does not suffer from this problem:
  - One thread indicates that a target thread is to be canceled,
  - The cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled.
  - The thread can perform this check at a point at which it can be canceled safely.

Pthread Cancellation

- Phread cancellation is initiated using the function:

  `pthread\_cancel()`

  The identifier of the target Pthread is passed as a parameter to the function.

- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Pthread Cancellation (Cont.)

- Invoking **`pthread_cancel()`** indicates only a request to cancel the target thread.
- The actual cancellation depends on how the target thread is set up to handle the request.
- Pthread supports three cancellation modes. Each mode is defined as a state and a type. A thread may set its cancellation state and type using an API.

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it

Thread Cancellation (Cont.)

- The default cancelation type is the deferred cancelation
  - Cancellation only occurs when thread reaches **cancellation point**
  - One way for establishing a cancellation point is to invoke the **`pthread_testcancel()`** function.
  - If a cancellation request is found to be pending, a function known as a **cleanup handler** is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated.
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

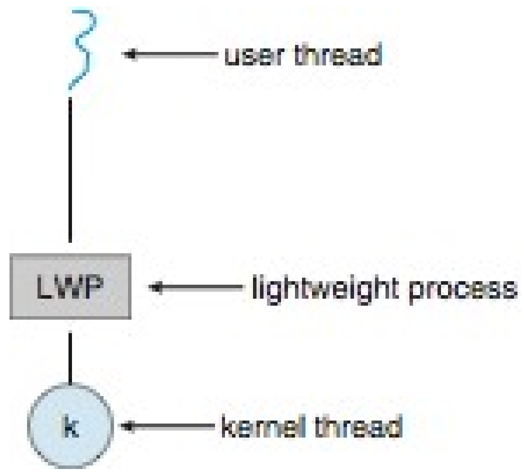
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

Scheduler Activations

- Both many-to-many and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.
- Typically uses an intermediate data structure between user and kernel threads
- This data structure is known as a – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP is attached to kernel thread.
  - The kernel threads are the ones that the operating system schedules to run on physical processors.

Scheduler Activations (Cont.)

- Lightweight process schema



- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads

Operating System Examples

- Windows Threads
- Linux Threads

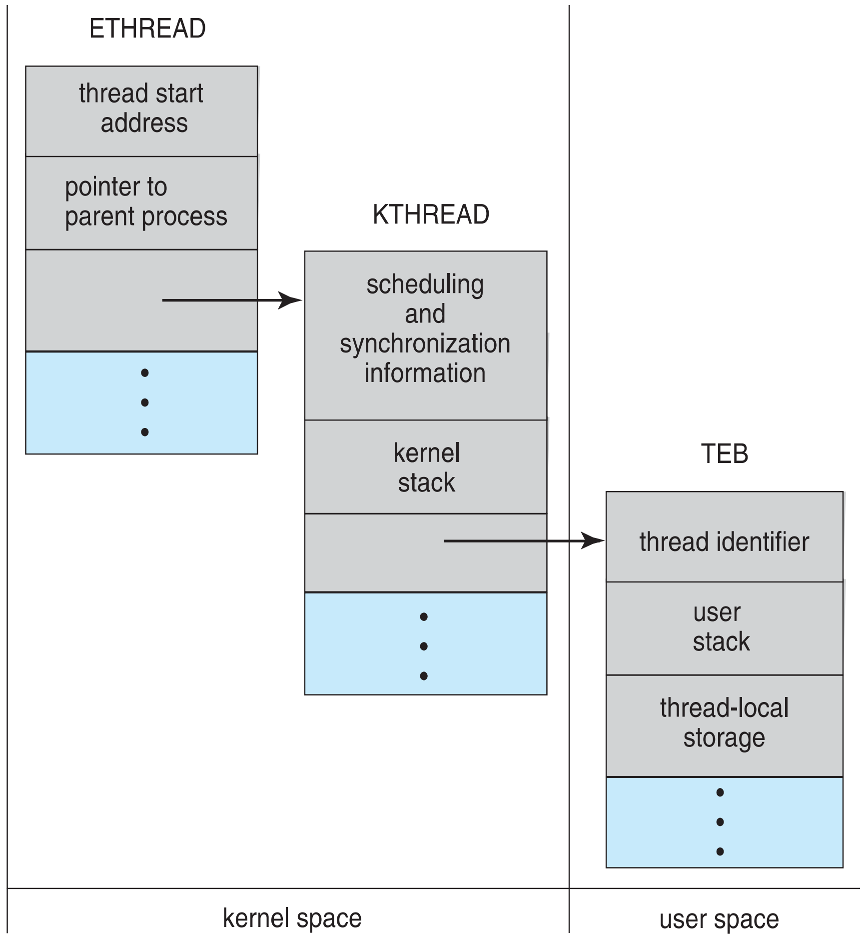
Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



Linux Threads

- Linux does not distinguish between processes and threads.
  - Linux uses the term **task** than process or thread when referring to a flow of control within a program.
- Linux provides the **`fork()`** system call with the traditional functionality of duplicating a process,
- Linux also provides the ability to create threads using the **`clone()`** system call.
- The system **`clone()`** system call allows a child task to share the address space of the parent task (process).

Linux Threads (Cont.)

- When **`clone()`** is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks.
- Flags control behavior

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 example, suppose that **`clone()`** is passed the flags CLONE_FS, CLONE_VM, CLONE_SIGHAND and CLONE_FILES.
  - The parent and child tasks will then share the same file-system information, the same memory space, the same signal handlers, and the same set of open files.

Linux Threads (Cont.)

- Using **`clone()`** with all the flags set (as in the previous slide) is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task.
- If none of these flags is set when **`clone()`** is invoked, then no sharing takes place, resulting in functionality similar to that provided by the **`fork()`** system call.
- **`struct task_struct`** points to process data structures (shared or unique)

End of Chapter 4