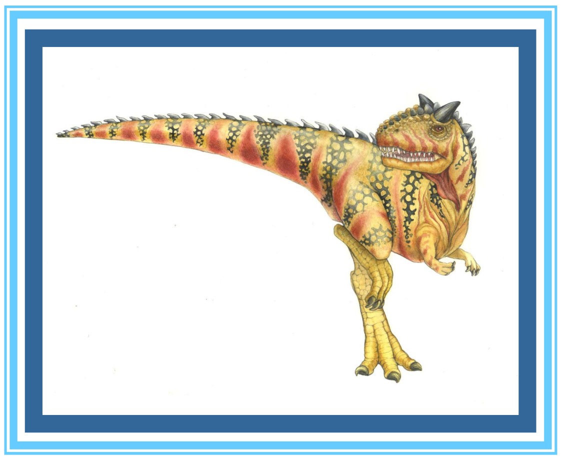
Chapter 4: Threads



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Threading Issues

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 examine issues related to **multithreaded programming**
- To cover operating **system support for threads** in Linux

Overview

- A thread is a basic unit of CPU utilization;
- it comprises
  - a **thread ID,**
  - **a program counter,**
  - **a register set, and**
  - **a stack**.
- It **shares with other threads** belonging to the same process its **code section, data section, and other operating-system resources**, such as open files and signals.
- A traditional (or heavyweight) process has **a single thread of control**.
- If a process has multiple threads of control, it **can perform more than one task at a time**.

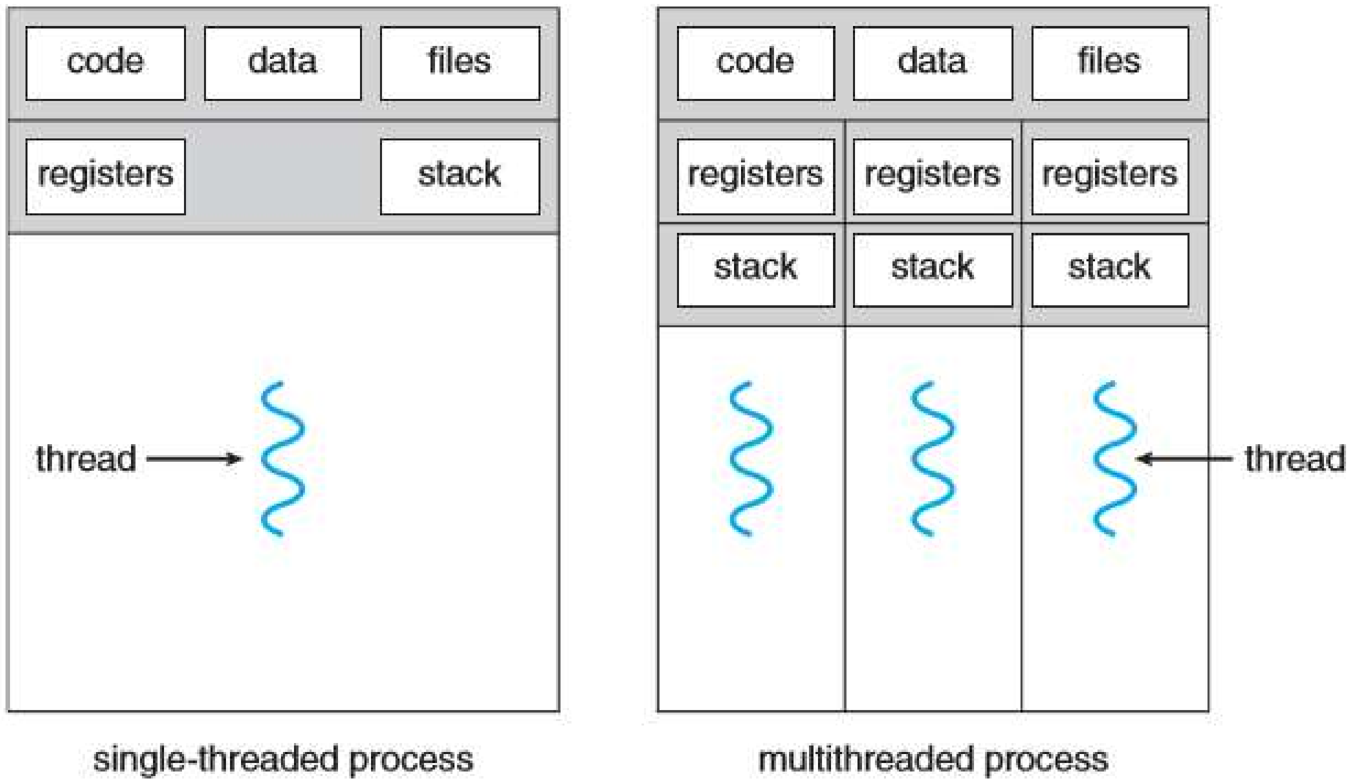
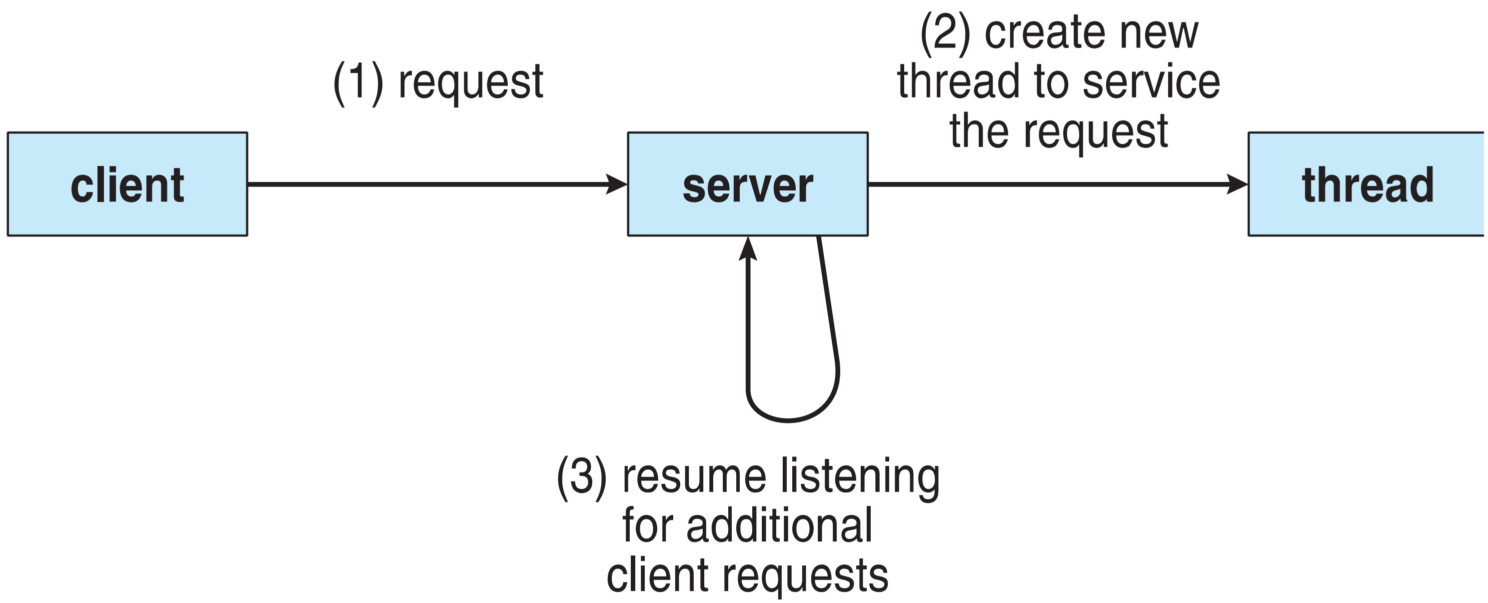


Figure 4.1 Single-threaded and multithreaded processes.

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 (time consuming and resource intensive) while thread creation is light-weight (because threads share the code, data and OS resources)
- If the web-server process is multithreaded, the server will create a separate thread that listens for client requests.
- When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.

Multithreaded Server Architecture



- If the web server ran as a traditional single-threaded process, it would be able to service only **one client at a time, and a client might have to wait a very long time** for its request to be serviced.

- Threads can simplify code, increase efficiency
- Kernels are generally multithreaded (Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling)

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
  - For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation.
  - A single-threaded application would be unresponsive to the user until the operation had completed.
  - In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.
- **Resource Sharing –**
  - **Processes can only share resources through techniques such as shared memory and message passing.**
  - **must be explicitly arranged by the programmer**
  - threads share resources of process, easier than shared memory or message passing

- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
  - because threads share the resources of the process to which they belong
- **Scalability –** process can take advantage of multiprocessor architectures
  - A single-threaded process can run on only one processor, regardless how many are available.

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, (to make better use of the multiple computing cores)
- challenges include:
  - **Dividing activities (in such a way that allow parallel execution of these activities)**
  - **Balance (programmers must also ensure that the tasks perform equal work of equal value)**
  - **Data splitting (the data accessed and manipulated by the tasks must be divided to run on separate cores)**
  - **Data dependency (when one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized (more on this in chap 5))**
  - **Testing and debugging (since many different execution paths are possible, testing and debugging such concurrent programs is inherently more difficult)**

- ***Parallelism*** implies a system can perform more than one task **simultaneously** (see Figure 4.4)
- ***Concurrency*** allows **more than one task to make progress**
  - Single processor / core, **scheduler** providing concurrency (see Figure 4.3)

Figure 4.3 Concurrent execution on a single-core system.

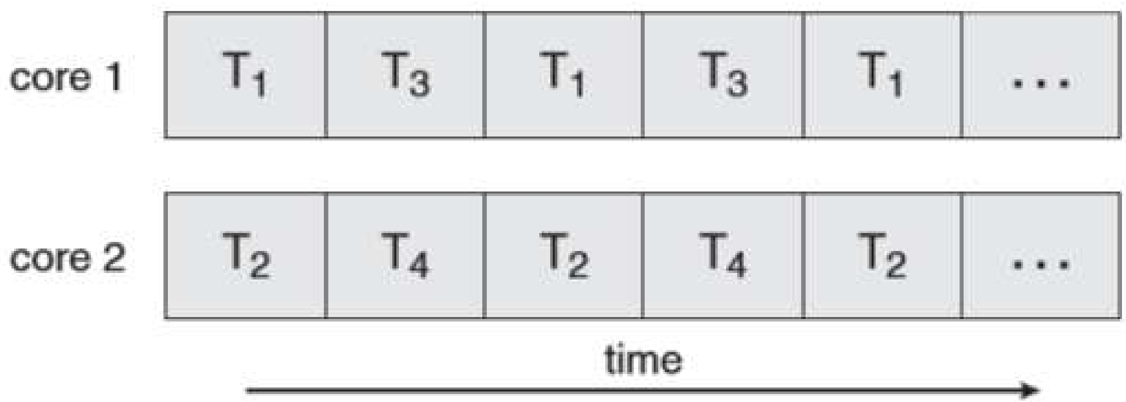


Figure 4.4 Parallel execution on a multicore system.

Multicore Programming (Cont.)

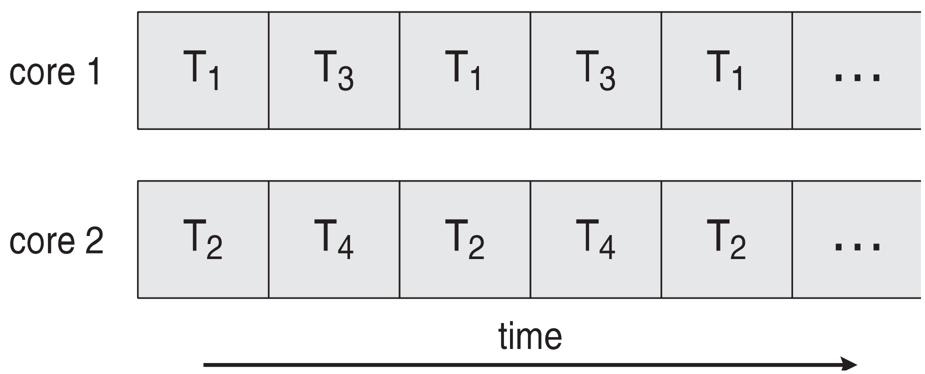
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - **Consider, for example, summing the contents of an array of size N. On a single-core system, one thread would simply sum the elements[0] ...[N−1].**
    - **On a dual-core system, however, thread A, running on core 0, could sum the elements [0] ...[N/2−1] while thread B, running on core 1, could sum the elements [N/2]...[N−1]. The two threads would be running in parallel on separate computing cores.**
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

- As # of threads grows, so does **architectural support for threading** **(for fast context switching)**
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

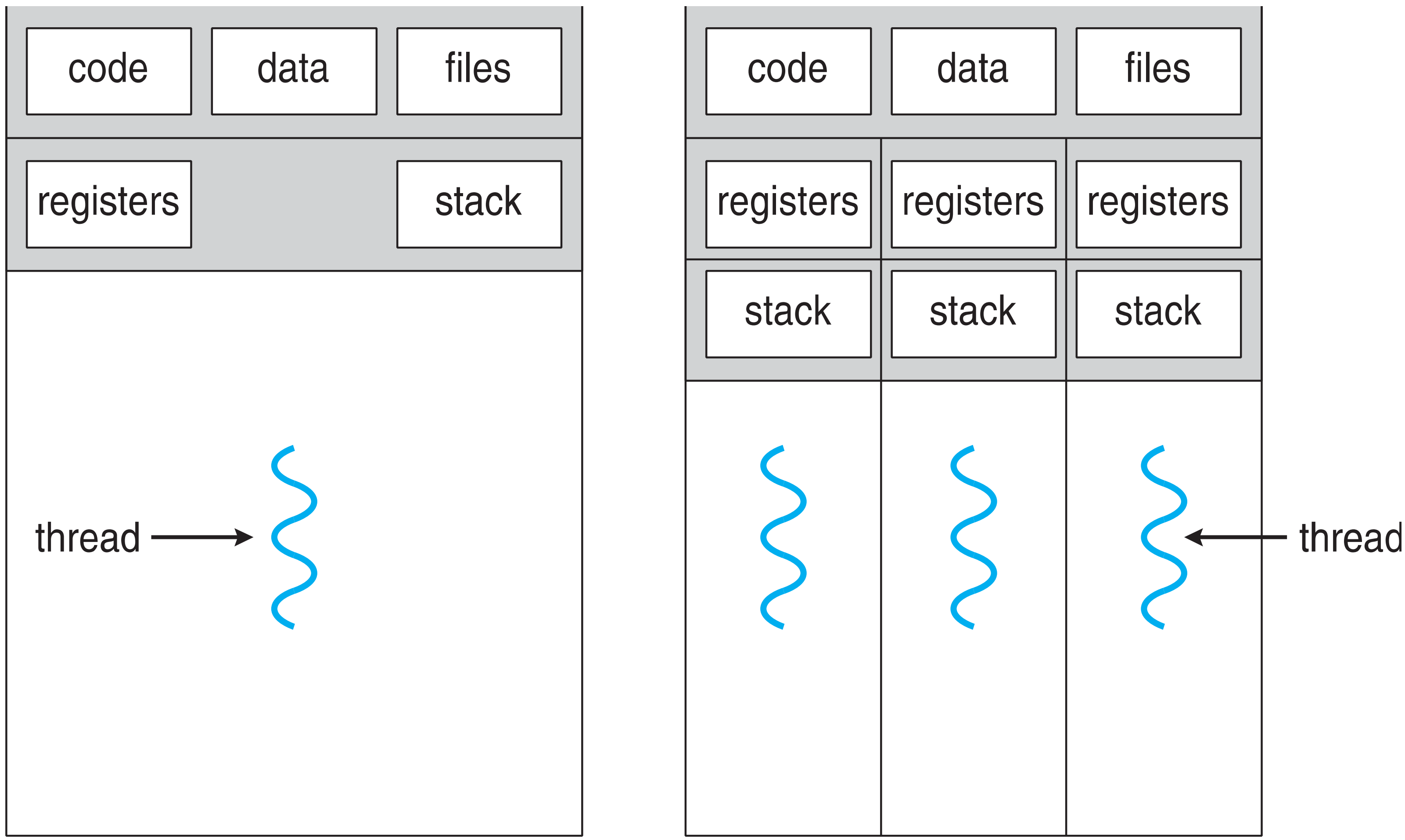
Concurrency vs. Parallelism

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

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



Single and Multithreaded Processes



single-threaded process

multithreaded process

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion (things that must be done sequentially)
- *N* processing cores

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

- That is, if application is 75% parallel / 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?

User Threads and Kernel Threads

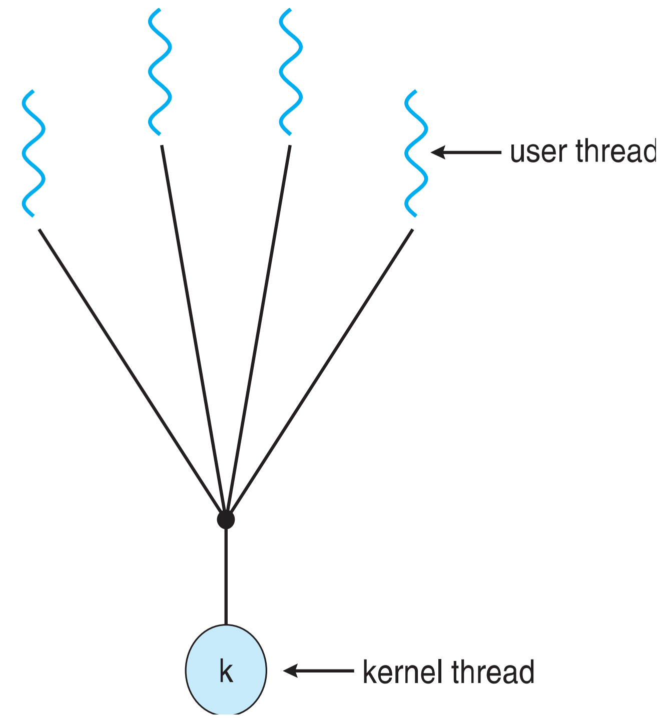
- **User threads** - management done by user-level threads library (without kernel support)
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel (supported and managed directly by the operating system)
- Examples – virtually all general purpose operating systems support kernel threads, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- Ultimately, a relationship must exist between user threads and kernel threads.
- Many-to-One
- One-to-One
- Many-to-Many

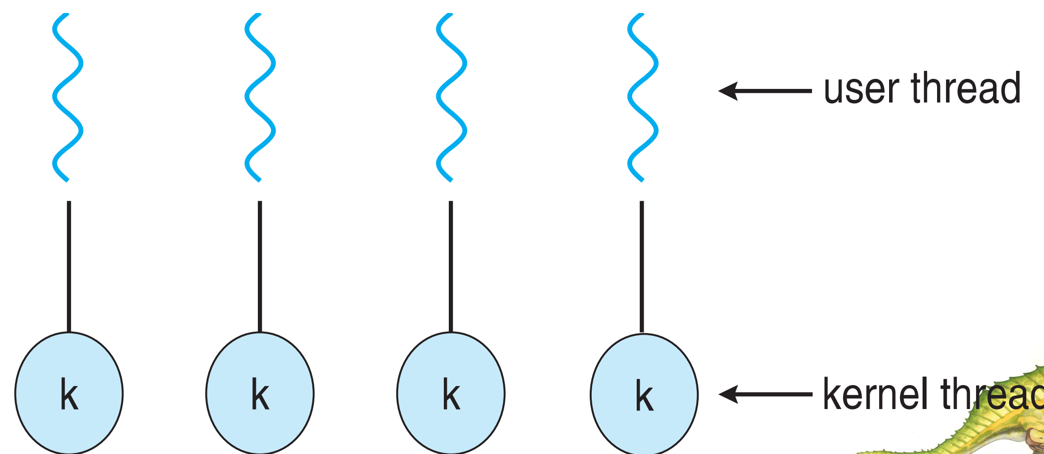
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads are **unable to run in parallel on muticore** system because only one may be in kernel at a time
- i.e. the kernel can schedule only one thread at a time
- Few systems currently use this model (because of its inability to take advantage of multiple processing cores)
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



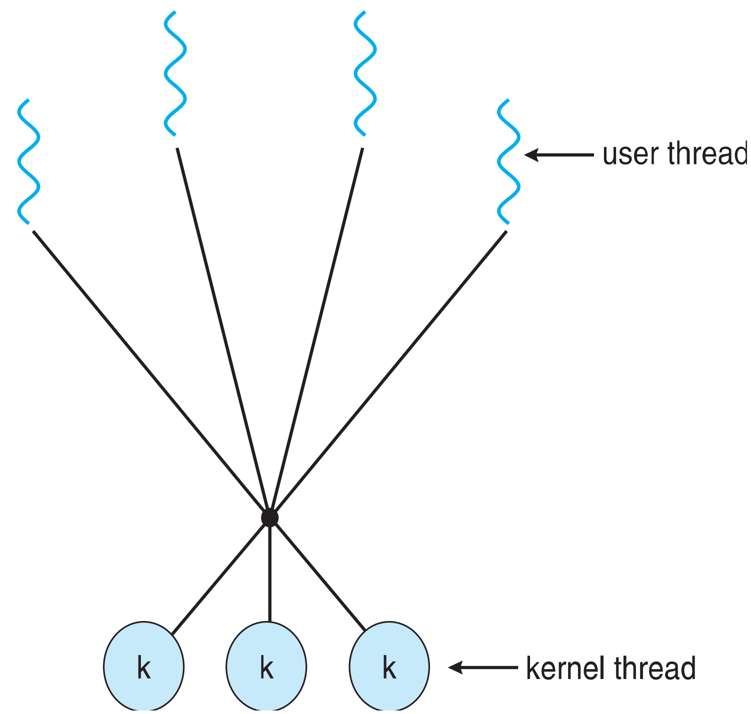
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- It also allows multiple threads to run in parallel on multiprocessors
- More concurrency than many-to-one
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



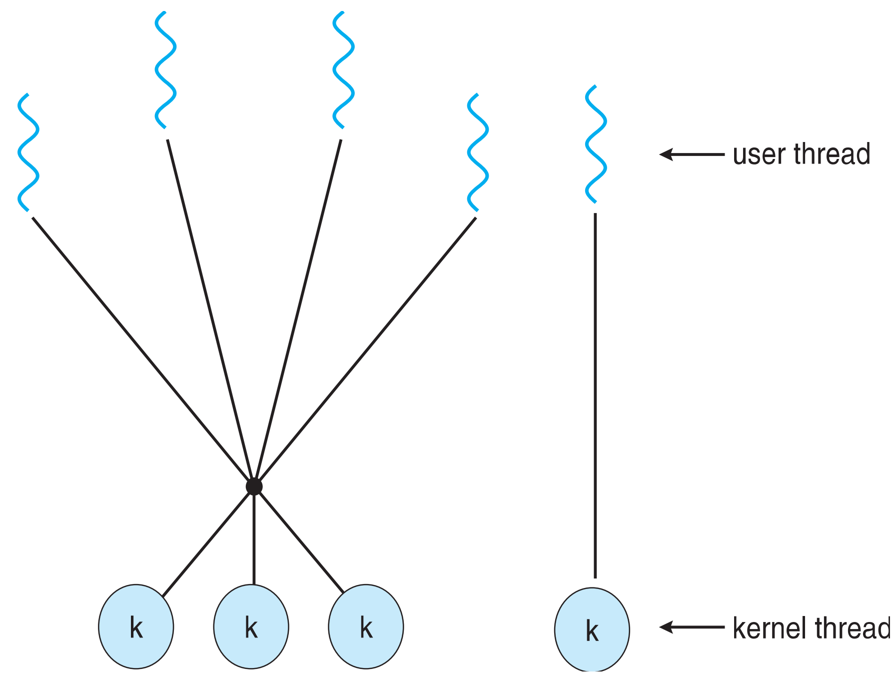
Many-to-Many Model

- Allows many user level threads to be mapped to a smaller or equal number of 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 M:M, 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
    - All code and data structures for the library exist in user space.
    - This means that invoking a function in the library results in a local function call in user space and **not a system call.**
  - Kernel-level library **supported directly by the OS**
    - Invoking a function in the API for the library typically results in **a system call to the kernel**.

- Three main thread libraries are in use today:
  - POSIX Pthreads
  - Win32 threads
  - Java threads.

Asynchronous threading and Synchronous threading

- Two general strategies for creating multiple threads:
- Asynchronous threading,
  - once the parent creates a child thread, **the parent resumes its execution**
  - the parent and child **execute concurrently**
  - The parent thread **need not know when its child terminates**
  - Typically **little data sharing** between threads
- Synchronous threading occurs when the parent thread creates one or more children and
  - then must wait for all of its children to terminate before it resumes
  - the so-called **fork-join strategy**.
  - Once each thread has finished its work**, it terminates and joins with its parent**
  - Only after all of the children have joined can **the parent resume execution**
  - Typically, synchronous threading **involves significant data sharing** among threads
  - the parent thread may **combine the results** calculated by its various children.
- **All of the following examples use synchronous threading**.

Pthreads

- May be provided **either as user-level or kernel-level**
- The threads extension of the POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - **POSIX stands for Portable Operating System Interface**, is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.
- ***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)
- **Any data declared globally**—that is, declared outside of any function—are **shared among all threads** belonging to the same process.

Example

- As an illustrative example, we design a multi threaded program that performs the summation of a non-negative integer in a separate thread

- For example, if N were 5, this function would represent the summation of integers from 0 to 5, which is 15.
- When this program begins, 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

```c
#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 10 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);
```

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Because **Java has no notion of global data**, access to **shared data must be explicitly arranged** between threads.
- Java threads may be created by:

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

  - Extending Thread class **or**
  - Implementing the Runnable interface

Creating Threads in Java

- There are 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—and more commonly used— technique is to define a class that implements the Runnable interface.
    - The code implementing the run() method is what runs as a separate thread.
    - Thread creation is performed by creating an object instance of the Thread class and passing the constructor a Runnable object.
    - The start() method creates the new thread
    - start() allocates memory and initializes a new thread in the JVM.
    - It also calls the run() method, making the thread eligible to be run by the JVM.
    - The join() method in Java is equivalent to pthread join() and WaitForSingleObject()

Sharing Data Between Threads in Java

- If two or more threads are to share data in a Java program, the sharing occurs by passing references to the shared object to the appropriate threads.
- In our example the main thread and the summation thread share the object instance of the Sum class.
- This shared object is referenced through the appropriate getSum() and setSum() methods

Java Multithreaded Program

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

Extra Example on Threads in Java

2. Creating threads

(1) Inheriting from the `Thread` class

- The general approach is
  - **(1) Define a class** by **extending** the `Thread` class and **overriding** the `run` method.
    - In the run method, you should write the code that you wish to run when this particular thread has started.
  - **(2) Create an instance** of the above class
  - **(3) Start running the instance** using the `start` method that is defined in `Thread`

```
public class WhereAmI extends Thread {
    int n;
// constructor
    public WhereAmI(int number) {
        n = number;
    }
// override the run method
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("I'm in thread " + n);

    }
}
```

```
public class ThreadTester {
    public static void main(String[] args) {
        // create the threads
        WhereAmI place1 = new WhereAmI(1);
        WhereAmI place2 = new WhereAmI(2);
        WhereAmI place3 = new WhereAmI(3);
        // start the threads
        place1.start();
        place2.start();
        place3.start();
    }
}
```

The output

```
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
```

2. Creating threads

(2) Implementing the `Runnable` interface

- The general approach is
  - (1) **Define a class** that **implements** `Runnable` and **overriding** the `run` method.
  - (2) **Create an instance** of the above class.
  - (3) **Create a thread** that runs this instance.
  - (4) **Start running the instance** using the `start` method.

```java
public class WhereAmI2 implements Runnable{
    int n;
    // constructor
    public WhereAmI2(int number) {
        n = number;
    }
    // override the run method
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("I'm in thread " + n);

    }
}
```

```java
public class ThreadTester2 {
    public static void main(String[] args) {
        // create a runnable objects,
        // and the thread to run them.
        WhereAmI2 place1 = new WhereAmI2(1);
        Thread thread1 = new Thread(place1);
        WhereAmI2 place2 = new WhereAmI2(2);
        Thread thread2 = new Thread(place2);
        WhereAmI2 place3 = new WhereAmI2(3);
        Thread thread3 = new Thread(place3);
        // start the threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

The output

```
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 1
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
I'm in thread 3
I'm in thread 2
```

Threading Issues

- In this section, we discuss some of the issues to consider in designing multithreaded programs

- 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()  (when invoked by a thread)`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- **`exec()`** usually works as normal – replace the running process including all threads.
  - That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire 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 kernel runs when handling signal
    - **User-defined signal handler** can override default
    - For single-threaded, signal delivered to process

Synchronous Signals

- A signal may be received either synchronously or asynchronously
- Examples of synchronous signal include
  - illegal memory access and
  - division by 0.
- If a running program performs either of these actions, a signal is generated.
- Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

Asynchronous Signals

- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
- Examples of such signals include terminating a process with specific keystrokes (such as **<control><C>)** and having a timer expire.
- Typically, an asynchronous signal is sent to another process.

Signal Handling (Cont.)

- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process.
- However, delivering signals is more complicated in multithreaded programs
- Where should a signal be delivered for multi-threaded?
  - 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 thread to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished
- For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further.
  - Often, a web page loads using several threads—each image is loaded in a separate thread.
  - When a user presses the stop button on the browser, all threads loading the page are canceled.

Thread Cancellation

- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately (may not free a necessary system-wide resource)
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled, allowing it an opportunity to terminate itself in an orderly fashion.
- 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);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state
- A thread cannot be canceled if cancellation is disabled

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

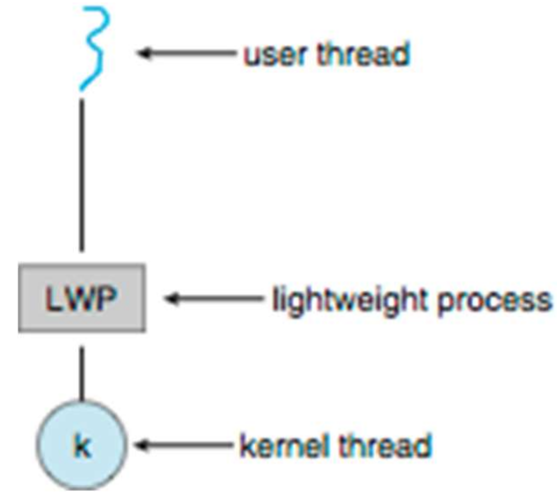
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-local storage.
- 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
- Most thread libraries—including Windows and Pthreads—provide some form of support for thread-local storage; Java providessupport as well

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- 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



End of Chapter 4