Chapter 4: Threads & Concurrency

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

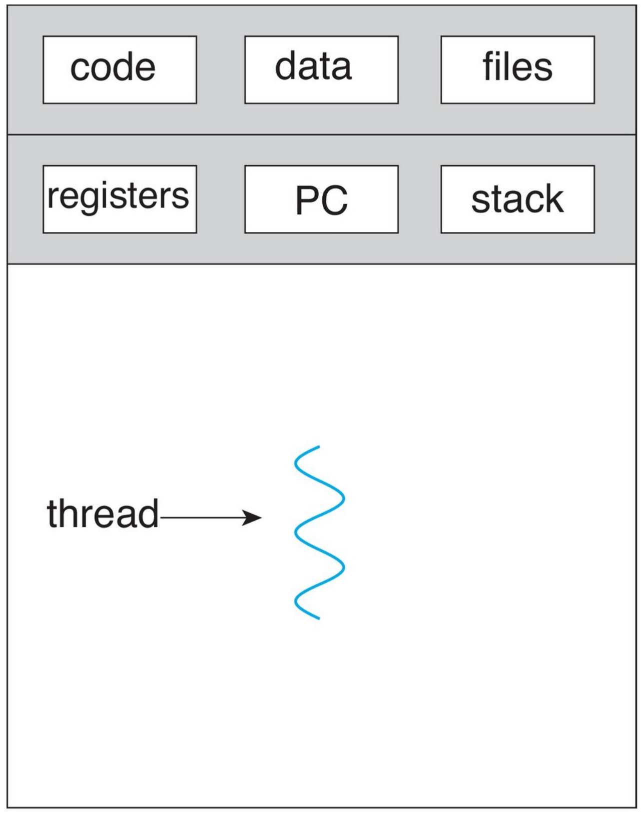
Overview

A thread is a **basic unit of CPU utilization**

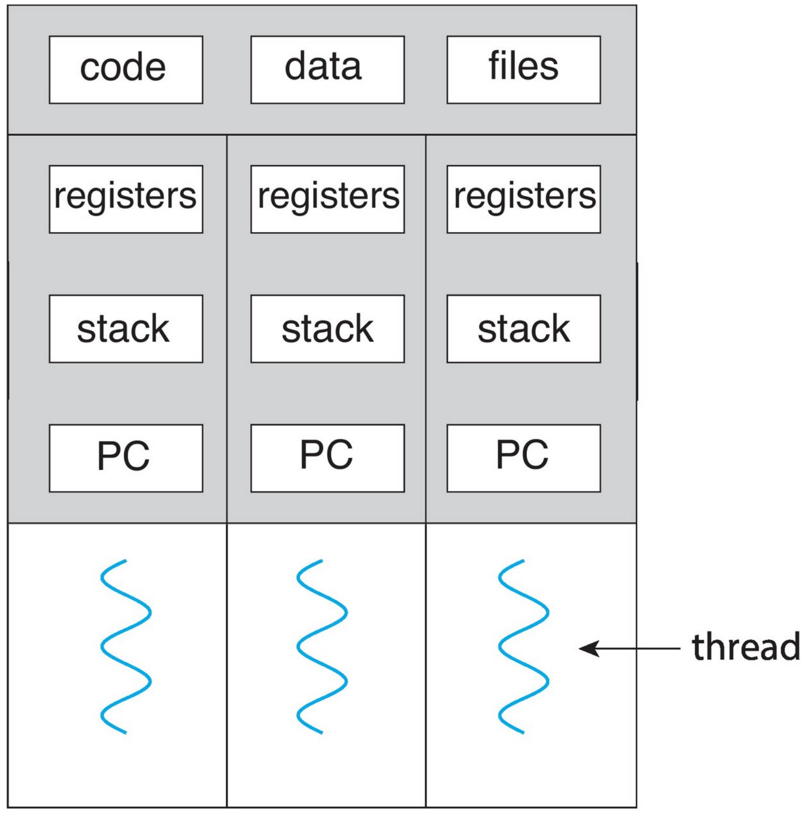
- It comprises a
  - thread ID, a
  - program counter (PC), a
  - register set, and a
  - stack.
- It **shares** with other threads belonging to the same process its
  - code section,
  - data section, and other
  - operating-system resources, such as open files and signals.

- A traditional 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.

Single and Multithreaded Processes



single-threaded process



multithreaded process

Motivation

- **Most modern applications are multithreaded**
- Threads run within application
- Process creation is time consuming, resource intensive, and hence heavy-weight. Thread creation is light-weight because all the threads within the process share the same address space, and there is no need to switch the address space.
- Can simplify code, increase efficiency
- Applications designed to leverage processing capabilities on multicore systems can perform several CPU-intensive tasks in parallel across the multiple computing cores.
- **Kernels are generally multithreaded**

Multithreaded applications

Multiple tasks with the application can be implemented by separate threads

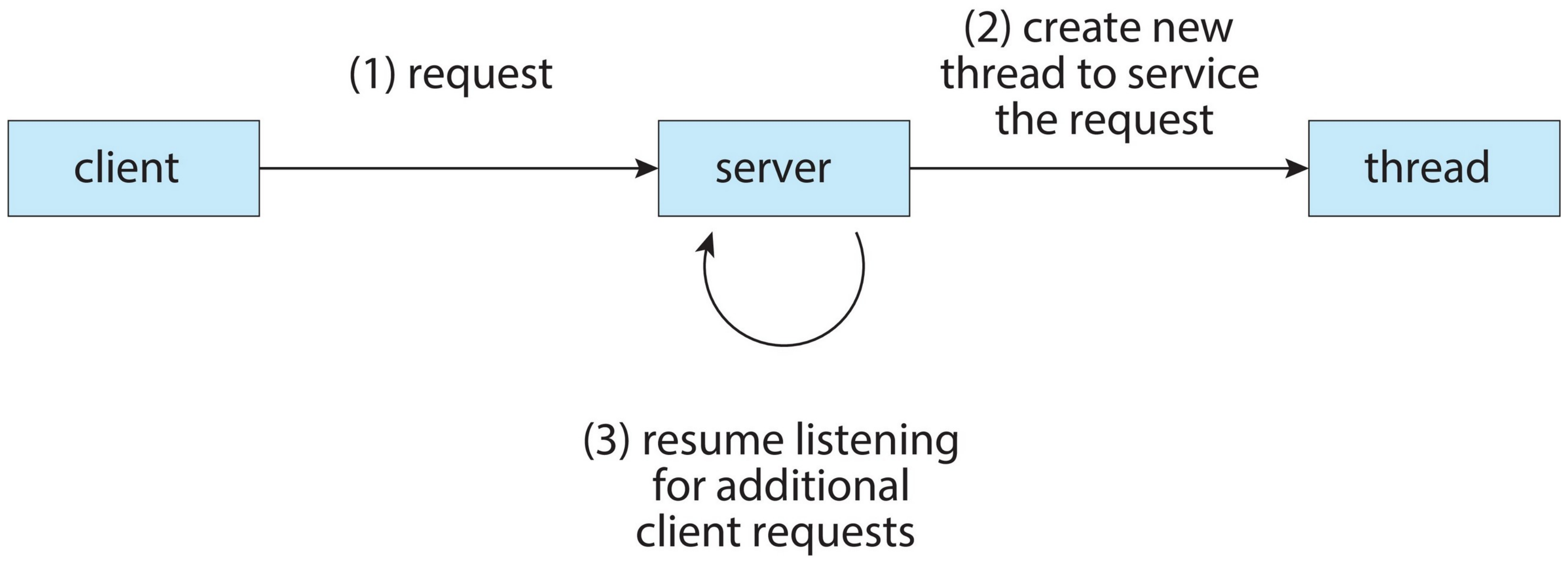
- A web browser might have
  - one thread display images or text while
  - another thread retrieves data from the network.

- A word processor may have
  - a thread for displaying graphics,
  - another thread for responding to keystrokes from the user, and
  - a third thread for performing spelling and grammar checking in the background.

A single application may be required to perform several similar tasks.

- It is generally more efficient to use one process that contains multiple threads.

Multithreaded Web Server


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

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.

If the web-server process is multithreaded, 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.

Benefits

- **Responsiveness** –Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource Sharing** – Threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

  (Whereas, processes share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer.)
- **Economy** – Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads than process creation.
- **Scalability** – The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.
  - A single-threaded process can run on only one processor, regardless how many are available.

4.2 Multicore Programming

Multicore Programming

Multicore systems – Multiple computing core on a single processing chip where each core appears as a separate CPU to the OS.

- **Multithreaded programming** provides a mechanism for more efficient use of these multiple computing cores and improved concurrency.

On a system with a single computing core,

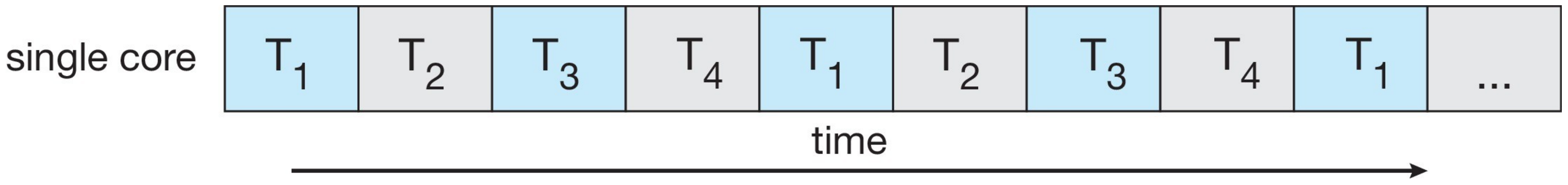
- concurrency merely means that the execution of the **threads** will be **interleaved over time** because the processing core is capable of executing **only one thread at a time**.
- Single processor / core, **scheduler providing concurrency**
- **it is possible to have concurrency without parallelism**.

On a system with multiple cores, however, concurrency

- means that the **threads** can **run in parallel**, because the system can assign **separate thread to each core**

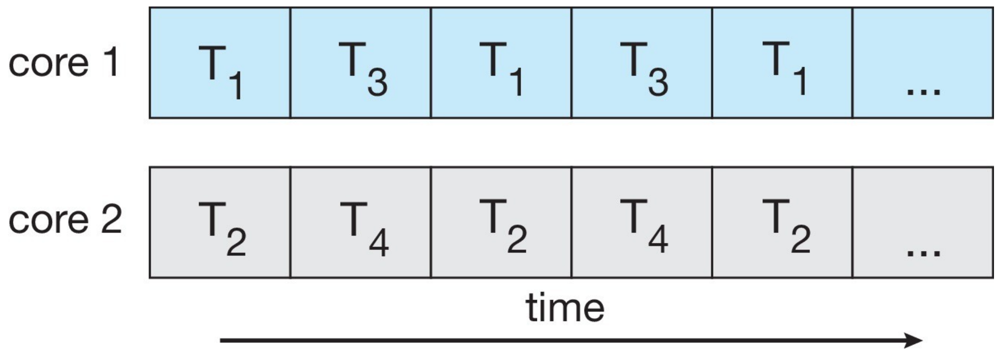
Concurrency vs. Parallelism

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



Concurrency supports more than one task making progress

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



Parallelism implies a system can perform more than one task simultaneously

Multicore Programming Challenges

- OS Designers must write scheduling algorithms that use multiple processing cores to allow the parallel execution.
- For Application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded

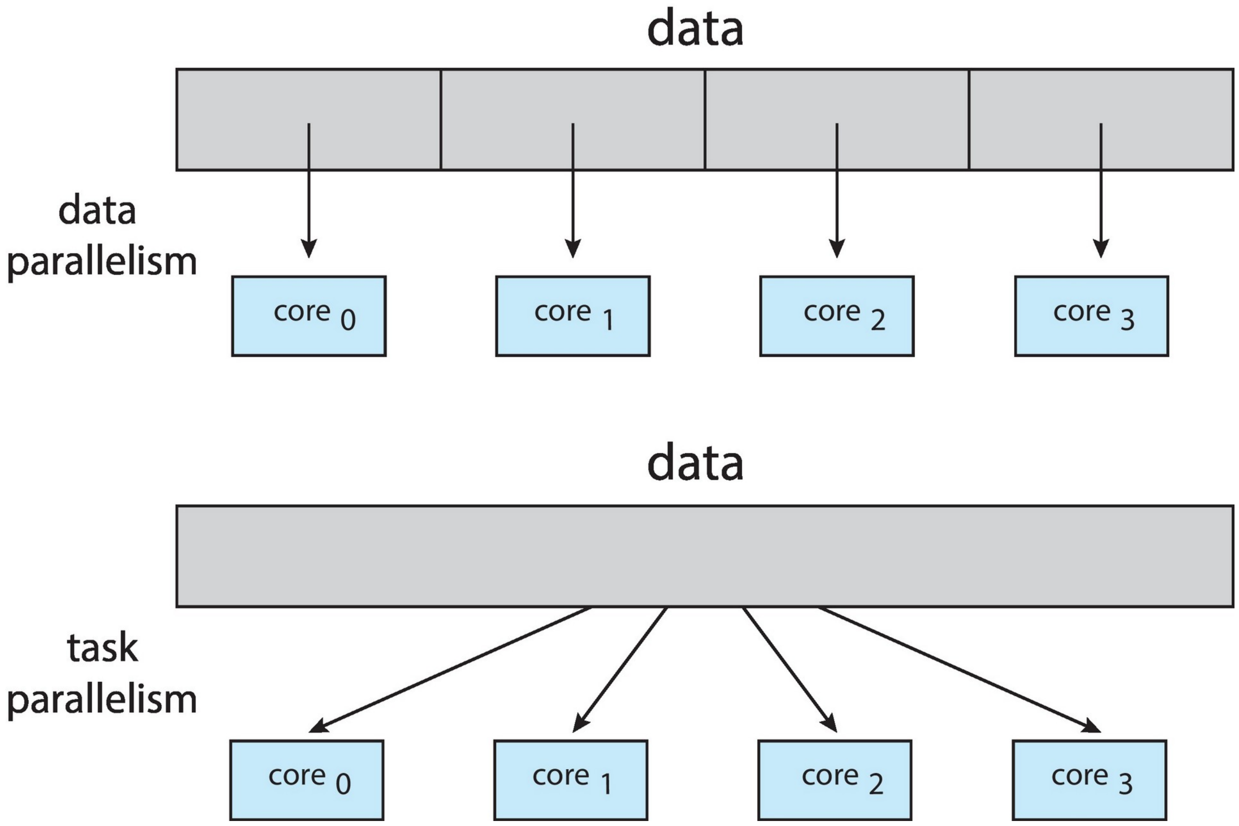
Challenges include:

1. **Dividing activities**
   - Examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
2. **Balance**
   - Ensure that the tasks perform equal work of equal value.
3. **Data splitting**
   - The data must be divided to run on separate cores.
4. **Data dependency**
   - When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized.
5. **Testing and debugging:** More difficult since many different execution paths

Types of Parallelism

- **Data parallelism** – distributes subsets of the same data across multiple cores,
- **Task parallelism** – distributing not data but tasks (threads) across multiple computing cores.

Not mutually exclusive, and an application may use a hybrid of these two strategies



Data parallelism

- **Data parallelism** focuses on **distributing subsets of the same data** across multiple computing cores and performing the **same operation on each core.**

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:

    Thread *A*, running on core 0, could sum the elements [0] . . . [*N*/2 − 1]

    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

Task parallelism involves **distributing 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.

Consider an example of task parallelism:

Two threads, each performing a unique statistical operation on the array of elements.

The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

Amdahl's Law

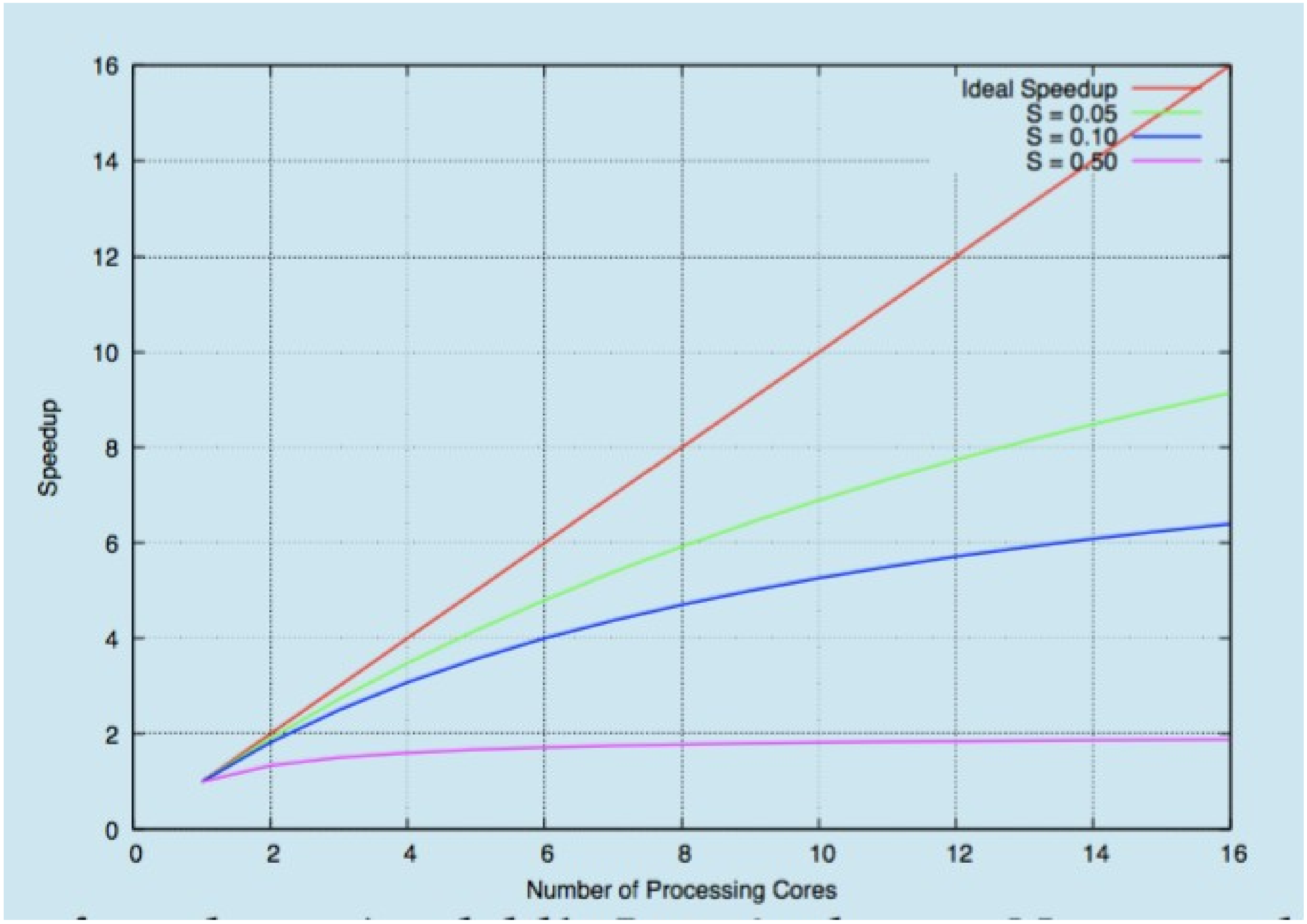
$$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$

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

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times.
  - 1/(0.25 + 0.75/2) = 1.6   // 25% serial, 75% parallel 2 cores
- If application is .05 serial, then parallel is 1- 0.05 = .95. Assume N = 8 cores.
  - 1/(0.05 + 0.95/8) = 5.9   // 5% serial, 95% parallel 8 cores
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Amdahl's Law


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

4.3 Multithreading Models

User level threads and Kernel Level Threads

A thread can be user level or kernel level depending on whether a user library or the OS kernel is managing it.

User level threads:

- User level threads are **managed by a user level library**.
- They still require a kernel system call to operate. It does not mean that the kernel knows anything about user level thread management. Kernel only takes care of the execution part.

Kernel Threads:

- Kernel level threads are **managed by the OS**, therefore, thread operations (such as Scheduling) are implemented in the kernel code.
- They are same as user space threads in many aspects, but one of the biggest difference is that they exist in the kernel space and execute in a privileged mode and have full access to the kernel data structures.
- These are basically used to implement background tasks inside the kernel. The task can be handling of asynchronous events or waiting for an event to occur.

User Threads and Kernel Threads

Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**.

User threads - management done by user-level threads library

- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

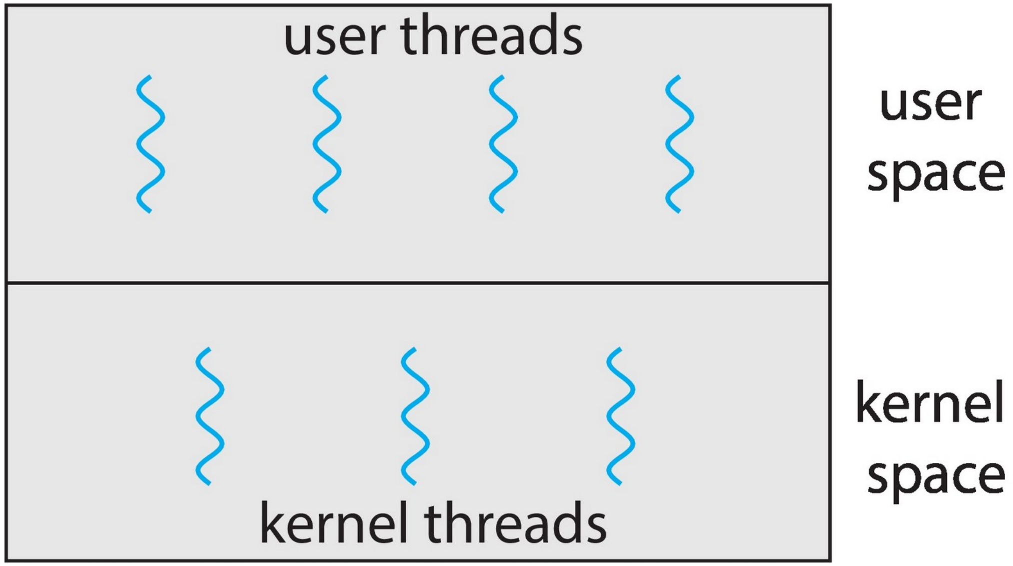
- **Kernel threads** - kernel threads are supported and managed directly by the operating system.

  How many threads are in a kernel? *cat /proc/sys/kernel/threads-max*

  - Supported by virtually all contemporary operating systems, including:
    - Windows
    - Linux
    - macOS
    - iOS
    - Android

4.3 Thread Libraries

User and Kernel Threads



A relationship exists between user threads and kernel threads, mapping user threads to kernel threads

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

- When we say "user-level threads **map to kernel threads**" we mean that the abstraction of **threads presented to user-space** is implemented using threads in kernel-space, with each user thread being **represented by a kernel-implemented thread**.

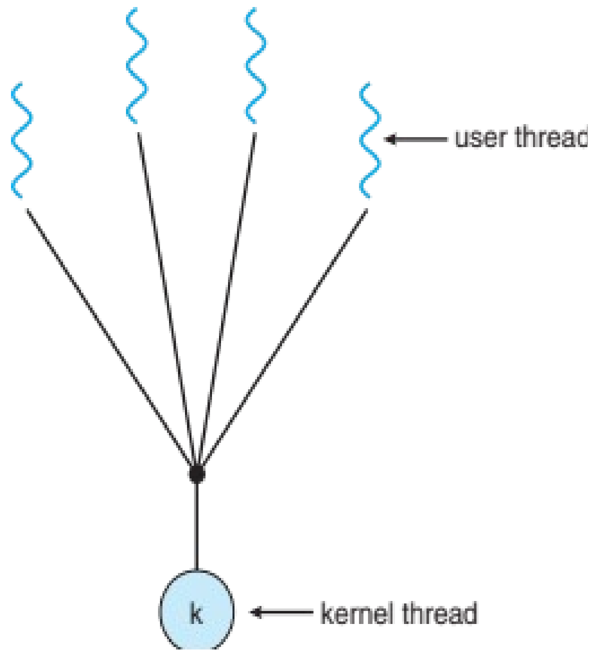
Many-to-One

- 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 because of its inability to take advantage of multiple processing cores
- Examples:
  - **Solaris Green Threads**

    a thread library available for

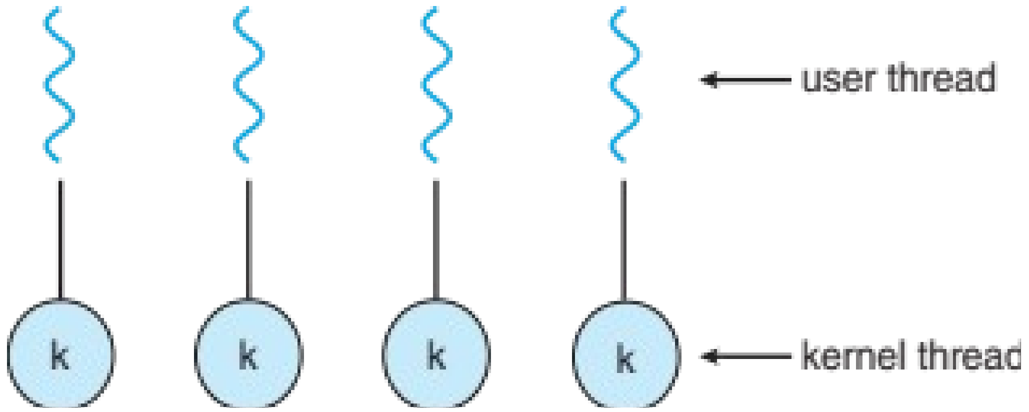
    Solaris systems and adopted in early

    versions of Java



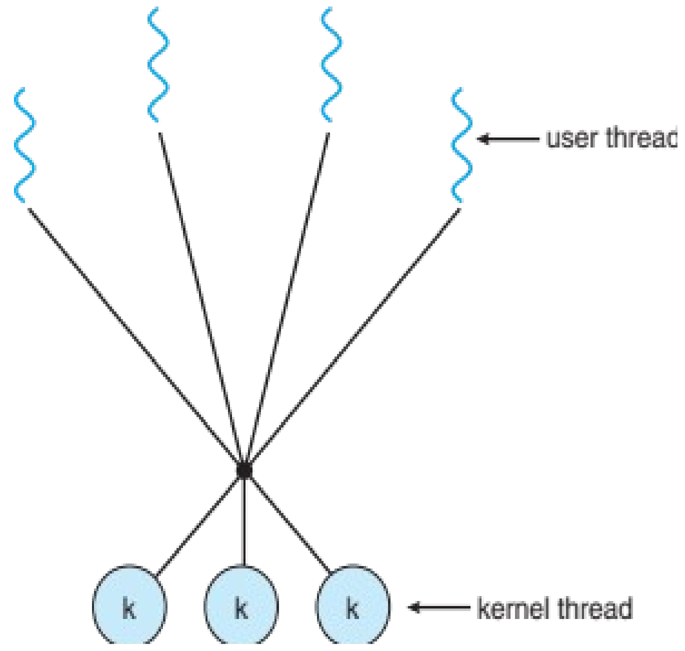
One-to-One

- Each user-level thread maps to kernel thread
- Drawback: creating a user-level thread requires creating a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



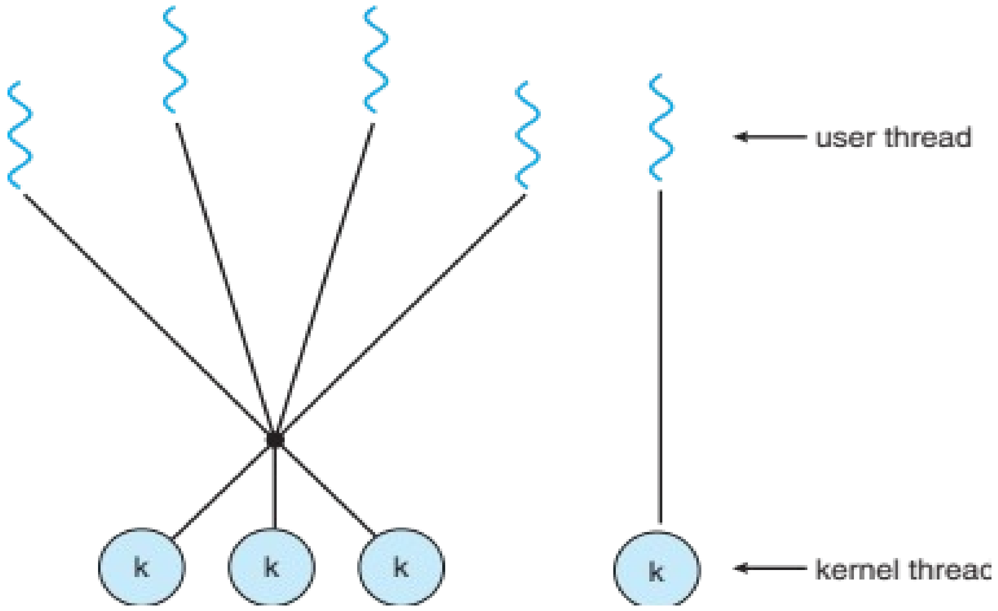
Many-to-Many Model

- Allows the operating system to create a sufficient number of kernel threads
- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores).
- Not very common



Two-level Model

- One variation on the many-to-many model still multiplexes many userlevel threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread.



4.4 Thread Libraries

Thread Libraries

- **Thread library** provides programmer with **API** for creating and managing threads
- Two primary ways of implementing
  1. **Library** entirely in **user space**
  2. **Kernel-level library** supported by the OS
     - Invoking a function in the API results in a **system call to the kernel.**
- 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
- ***POSIX is Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX, Linux & macOS systems

Pthreads, Windows threads, Java threads

1. **Pthreads,** the threads extension of the POSIX standard, may be provided as either a **user-leve**l or a **kernel-level library**.

2. The **Windows** thread library is a **kernel-level library** available on Windows systems.

3. The **Java thread API** allows threads to be created and managed directly in Java programs.
   - However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally **implemented** using a **thread library** available on the **host system**.
   - This means that on Windows systems, Java threads are typically implemented using the Windows API; UNIX, Linux, and macOS systems typically use Pthreads.

Global Data

For **POSIX and Windows** threading, any data declared globally—that is, declared outside of any function—are shared among all threads belonging to the same process.

As a pure object-oriented language, **Java has no** notion of **global data**. Access to shared data must be explicitly arranged between threads.

Data declared local to a function are typically stored on the stack. Since each thread has its own stack, each thread has its own copy of local data.

Asynchronous Threading

Two general strategies for creating multiple threads:

- **Asynchronous**
  - Once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently.
  - Each thread runs independently of every other thread, and the parent thread need not know when its child terminates.
  - Because the threads are independent, there is typically little data sharing between threads.
  - Asynchronous threading is the strategy used in the multithreaded server illustrated in the Multithreaded Web Server (slide 8)

Synchronous Threading

- 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.
  - Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed.
  - 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.
  - For example, the parent thread may combine the results calculated by its various children. **All** of the **following examples use synchronous threading**

Example of thread creation

- As an illustrative example, we design a multithreaded program that performs the summation of a non-negative integer in a separate thread using the well-known summation function:

$$sum = \sum_{i=0}^{N} i$$

- Thus, if the user enters 5 on the command line, the summation of the integer values from 0 to 5, which is 15, will be output

Pthreads Example

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

#include <stdlib.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 */

   /* set the default attributes of the thread */
   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);
}
```

Pthreads Example (cont)

```
/* The thread will execute 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);
```

Program explanation

1. All Pthreads programs must include the pthread.h header file.
2. The statement pthread_t tid declares the identifier for the thread we will create.
3. Each thread has a set of attributes, including stack size and scheduling information. The pthread attr_t_attr declaration represents the attributes for the thread.
4. We set the attributes in the function call pthread_attr_init(&attr).
5. Because we did not explicitly set any attributes, we use the default attributes provided.
6. A separate thread is created with the pthread_create() function call.
7. In addition to passing the thread identifier and the attributes for the thread, we also pass the name of the function where the new thread will begin execution in this case, the runner() function.
8. Last, we pass the integer parameter that was provided on the

Program explanation (Cont.)

9. At this point, the program has two threads: the initial (or parent) thread in main() and the summation (or child) thread performing the summation operation in the runner() function.
10. This program follows the thread create/join strategy, whereby after creating the summation thread, the parent thread will wait for it to terminate by calling the pthread_join() function.
11. The summation thread will terminate when it calls the function pthread_exit().
12. Once the summation thread has returned, the parent thread will output the value of the shared data sum.

Java Threads

Threads are the fundamental model of program execution in a Java program

- Java language and its API provide a rich set of features for the creation and management of threads.
- All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM.
- Java threads are available on any system that provides a JVM including Windows, Linux, and macOS. The Java thread API is available for Android applications as well

Java Threads (cont)

There are **two** techniques for explicitly creating threads in a Java program.

- Method 1:

  One approach is to create a new class that is **derive**d from the **Thread class** and to **override its run()** method.

- Method 2:

  An alternative—and more commonly used—technique is to define a class that **implements** the **Runnable interface.** This interface defines a single abstract method with the signature **`public void run()`**

Java Threads – example code

Implementing Runnable interface:

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

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Implementing Runnable interface: The code in the run() method of a class that implements Runnable is what executes in a separate thread.

Creating a thread involves creating a Thread object and passing it an instance of a class that implements Runnable, and invoking the start() method on the Thread object.
Note again that we never call the run() method directly. Rather, we call the start() method, and it calls the run() method on our behalf.

Java Executor Framework

Java introduced several new concurrency features that provide developers with much greater control over thread creation and communication. These tools are available in the java.util.concurrent package.

- Rather than explicitly creating Thread objects, thread creation is instead organized around the Executor interface:

```
public interface Executor
{
      void execute(Runnable command);
}
```

- Classes implementing this interface must define the execute() method which is passed a Runnable object.

```
Executor service = new Executor;
service.execute(new Task());
```

Thread Data Sharing

- Data sharing between threads belonging to the same process occurs easily in **Windows and Pthreads**, since **shared data** are simply declared **globally**.

- As a pure object-oriented language, Java has no such notion of global data. We can pass parameters to a class that implements Runnable, but Java threads cannot return results.

  - To address this need, the java.util.concurrent package additionally defines the Callable interface, which behaves similarly to Runnable except that a result can be returned.

  - Results returned from Callable tasks are known as Future objects. A result can be retrieved from the get() method defined in the Future interface. The program shown next illustrates the summation program using these Java features:

Java Executor Framework

The Summation class implements the Callable interface, which specifies the method call()
It is the code in this call() method that is executed in a separate thread.

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

class Summation implements Callable<Integer>
{
   private int upper;
   public Summation(int upper) {
      this.upper = upper;
   }

   /* The thread will execute in this method */
   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;

      return new Integer(sum);
   }
}
```

Java Executor Framework (cont)

```
public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```

4.5 Implicit Threading

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
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

Thread Pools

The general idea behind a thread pool is to create a number of threads at start-up and place them into a pool, where they sit and wait for work.

- When a server receives a request, rather than creating a thread, it instead submits the request to the thread pool and resumes waiting for additional requests.
- If there is an available thread in the pool, it is awakened, and the request is serviced immediately.
- If the pool contains no available thread, the task is queued until one becomes free.
- Once a thread completes its service, it returns to the pool and awaits more work.
- Thread pools work well when the tasks submitted to the pool can be executed asynchronously.

Thread Pools (Cont.)

Advantages:

1. Servicing a request with an existing thread is often faster than waiting to create a thread.
2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.

Java Thread Pools

- Three factory methods for creating thread pools in Executors class:

```
• static ExecutorService newSingleThreadExecutor()
• static ExecutorService newFixedThreadPool(int size)
• static ExecutorService newCachedThreadPool()
```

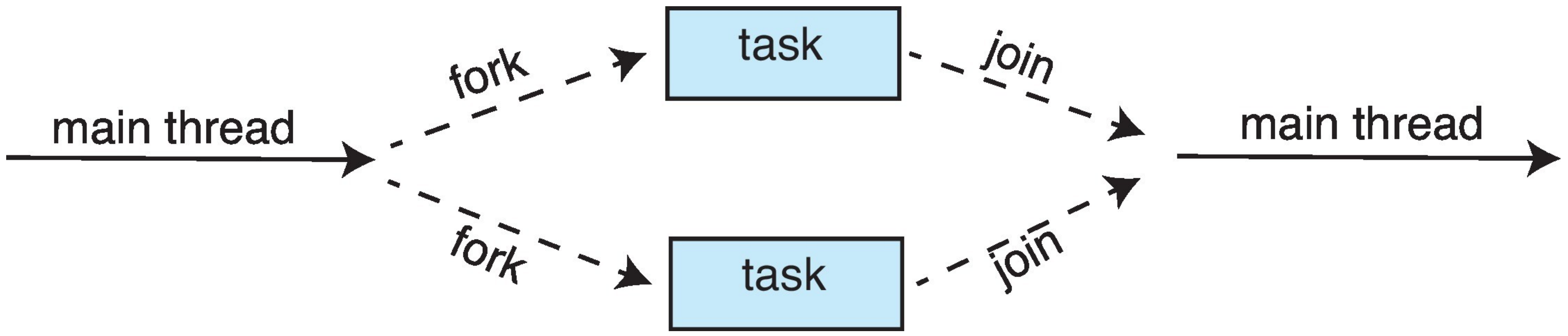
```
import java.util.concurrent.*;

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();
}
```

The example shown in Figure creates a cached thread pool and submits tasks to be executed by a thread in the pool using the execute() method

When the shutdown() method is invoked, the thread pool rejects additional tasks and shuts down once all existing tasks

Fork-Join Parallelism

- In explicit **fork-join** model, the main parent thread creates (***forks***) one or more child threads and then waits for the children to terminate and ***join*** with it, at which point it can retrieve and combine their results.
- In **implicit threading**, threads are **not constructed directly during the fork stage; rather, parallel tasks are designated**
- **A library manages** the number of threads that are created and is also responsible for assigning tasks to threads.
- Fork-join is a synchronous version of thread pools in which library determines the actual number of threads to create



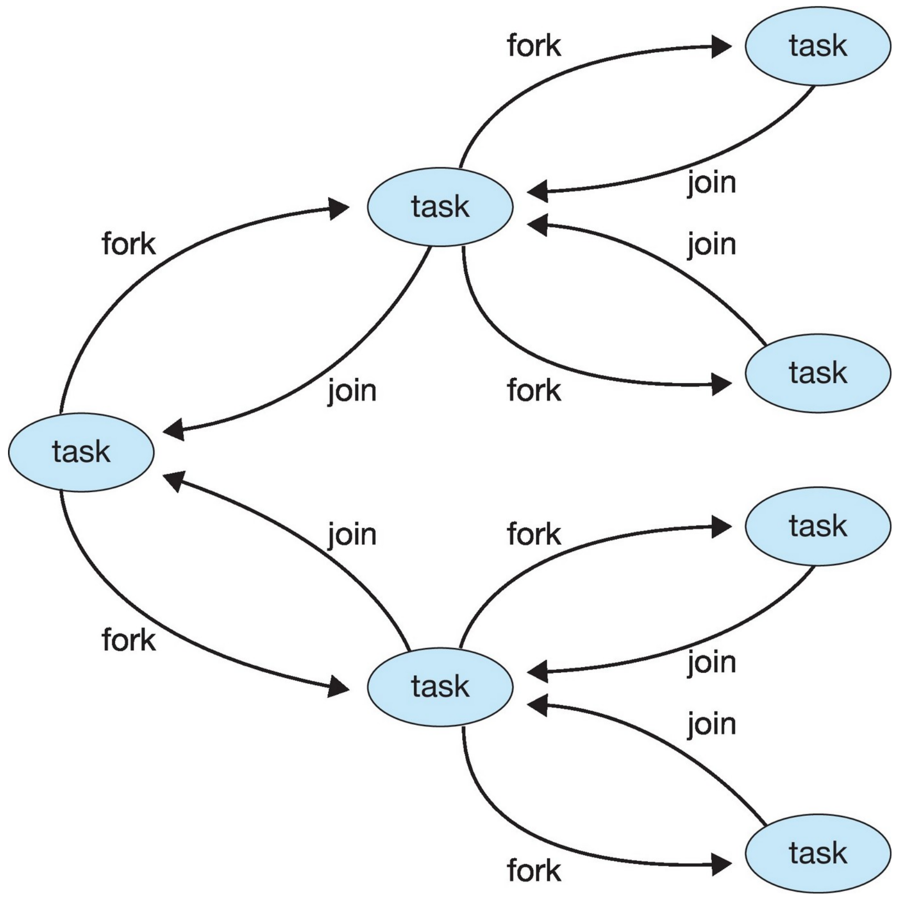
Fork-Join Parallelism in Java

Java introduced a fork-join library. When implementing divide-and-conquer algorithms using this library, separate tasks are forked during the divide step and assigned smaller subsets of the original problem.

- Algorithms must be designed so that these separate tasks can execute concurrently.
- At some point, the size of the problem assigned to a task is small enough that it can be solved directly and requires creating no additional tasks.
- The general recursive algorithm behind Java's fork-join model:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

Fork-Join Parallelism in Java (Cont.)



Fork-Join Calculation using the Java API

Recursive divide-and conquer algorithm that sums all elements in an array of integers

```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

Fork-Join Calculation using the Java API

```java
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;
   }

   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();
      }
   }
}
```

When the problem is “small enough” to be solved directly and no longer requires creating additional tasks:
In SumTask, this occurs when the number of elements being summed is less than the value THRESHOLD, which we have arbitrarily set to 1,000.

OpenMP

- OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared memory environments.
- Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP runtime library to execute the region in parallel
- **parallel regions** – blocks of code that can run in parallel
- When OpenMP encounters the directive

  `#pragma omp parallel`

  it creates as many threads as there are processing cores in the system

```
#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;
}
```

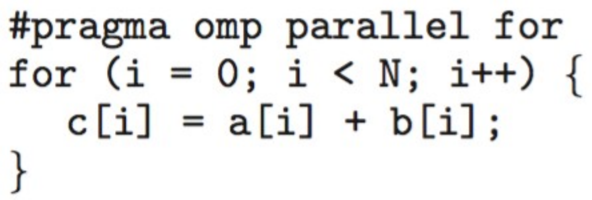
OpenMP (Cont.)

- For a dual-core system, two threads are created; for a quad-core system, four are created; and so forth.
- All the threads then simultaneously execute the parallel region.
- As each thread exits the parallel region, it is terminated.

OpenMP allows developers to choose among several levels of parallelism.
For example, they can set the number of threads manually. It also allows developers to identify whether data are shared between threads or are private to a thread

OMP Parallel Loop

Run the for loop in parallel



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

Grand Central Dispatch (GCD)

- Apple technology for macOS and iOS operating systems
- Extensions to C, C++ and Objective-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
    - Four system wide queues divided by quality of service:
    - `QOS_CLASS_USER_INTERACTIVE`
    - `QOS_CLASS_USER_INITIATED`
    - `QOS_CLASS_USER_UTILITY`
    - `QOS_CLASS_USER_BACKGROUND`

Grand Central Dispatch

- For the Swift language a task is defined as a closure – similar to a block, minus the caret
- Closures are submitted to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue
   (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```

Intel Threading Building Blocks (TBB)

- Intel threading building blocks (TBB) is a template library that supports designing parallel applications in C++.
- As this is a library, it requires no special compiler or language support. Developers specify tasks that can run in parallel, and the TBB task scheduler maps these tasks onto underlying threads.
- Furthermore, the task scheduler provides load balancing and is cache aware, meaning that it will give precedence to tasks that likely have their data stored in cache memory and thus will execute more quickly.
- TBB provides a rich set of features, including templates for parallel loop structures, atomic operations, and mutual exclusion locking

A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

- The same for loop written using TBB with **`parallel_for`** statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

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 UNIXes 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 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 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
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- 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);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

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 Cancellation in Java

- Deferred cancellation uses the **`interrupt()`** method, which sets the interrupted status of a thread.

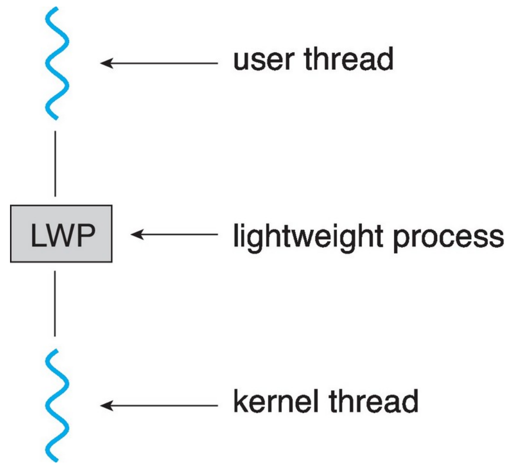
```
Thread worker;

  . . .

/* set the interruption status of the thread */
worker.interrupt()
```

- A thread can then check to see if it has been interrupted:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```

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



Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Linux also provides the ability to create threads **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - 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.

- **`struct task_struct`** points to process data structures (shared or unique)

End of Chapter 4