

Threads & Concurrency

Course Code: CSC 2209

Course Title: Operating Systems



Dept. of Computer Science
Faculty of Science and Technology

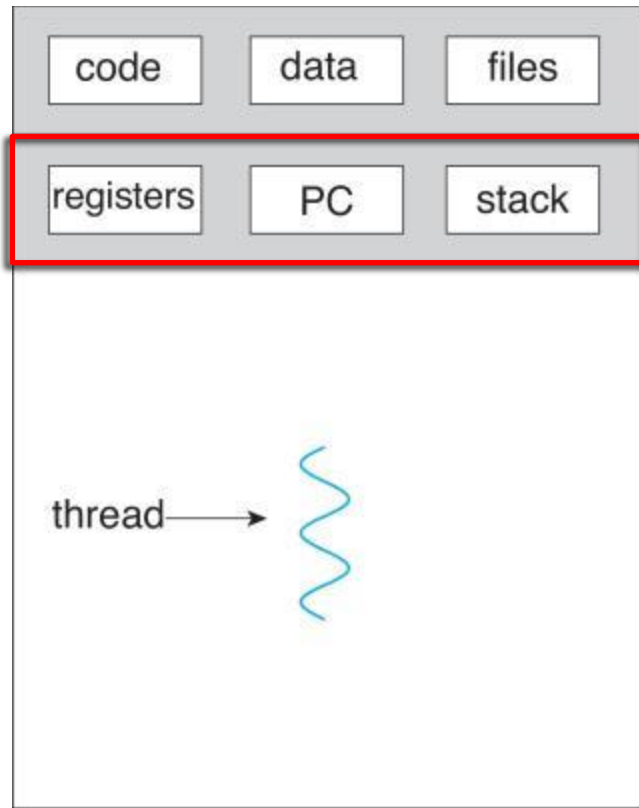
Lecturer No:	05	Week No:	05	Semester:	
Lecturer:	<i>Name & email</i>				

Lecture Outline

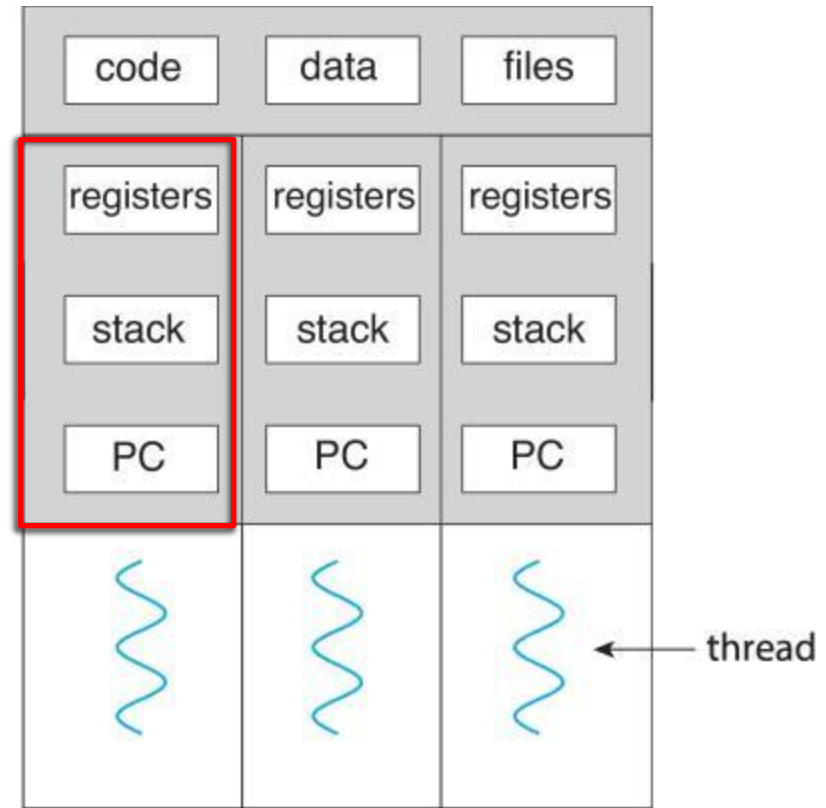


1. Overview
2. Multicore Programming *
3. Multithreading Models *
4. Thread Libraries
5. Implicit Threading
6. Threading Issues *
7. Operating System Examples

Single and Multithreaded Processes

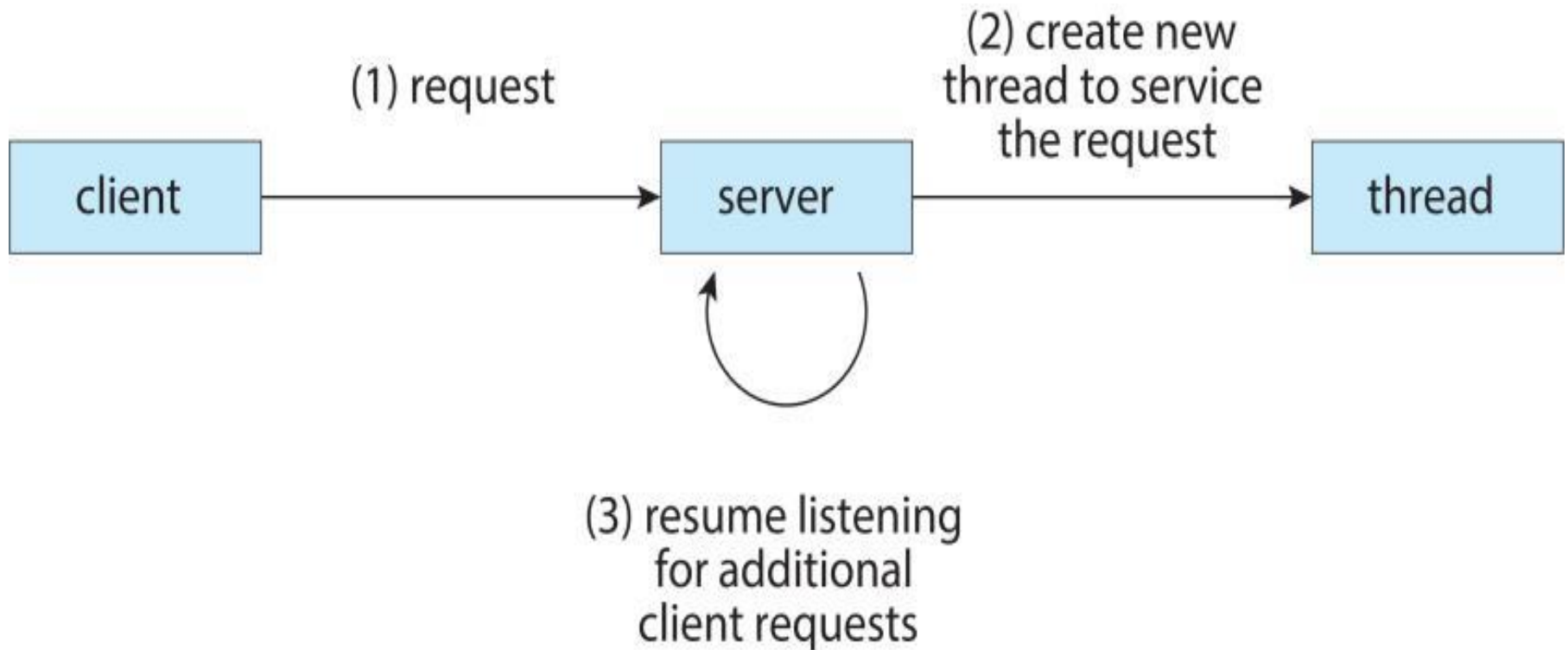


single-threaded process



multithreaded process

Multithreaded Server Architecture

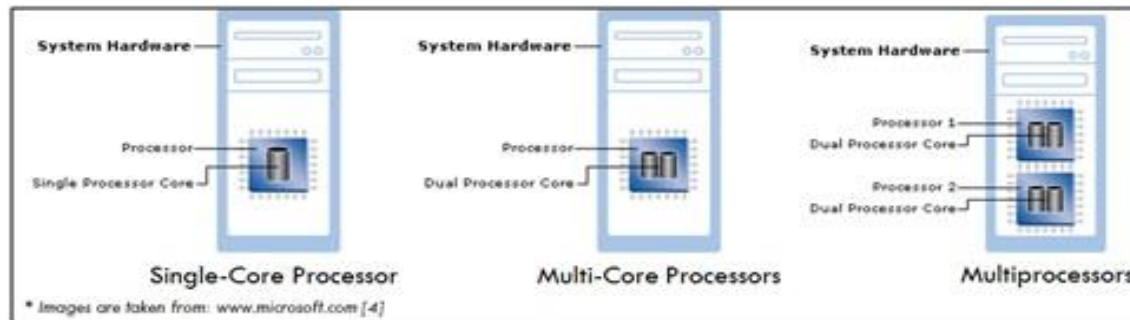


Benefits Multithreading (MT)

- ❑ **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 multicore architectures (utilization of Multiple Processor)

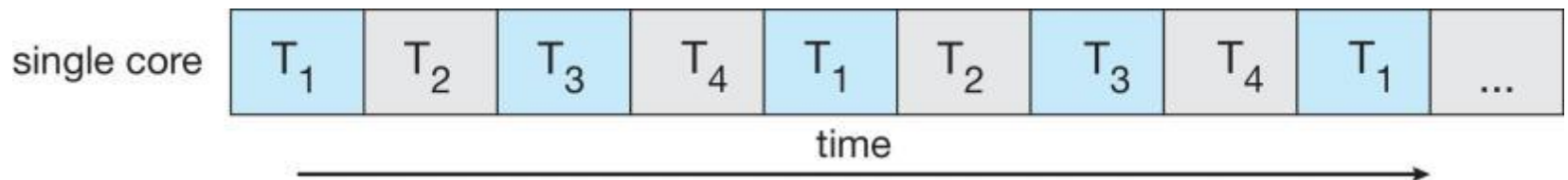
Multicore Programming

- ❑ **Multicore** or **multiprocessor** systems putting 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

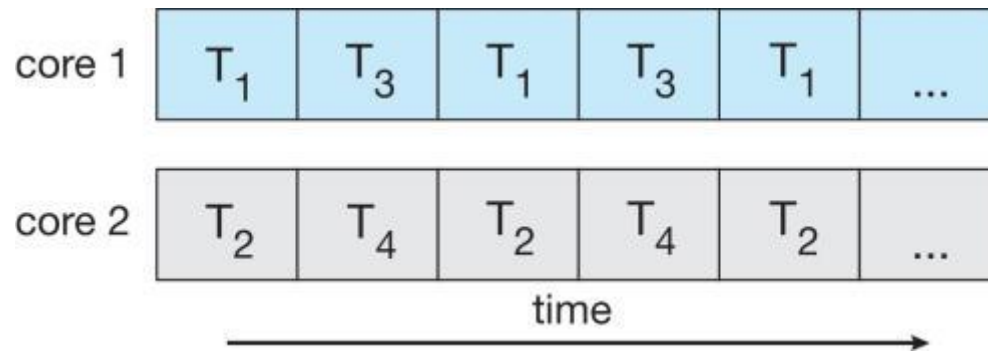


Concurrency vs. Parallelism

❑ Concurrent execution on single-core system:



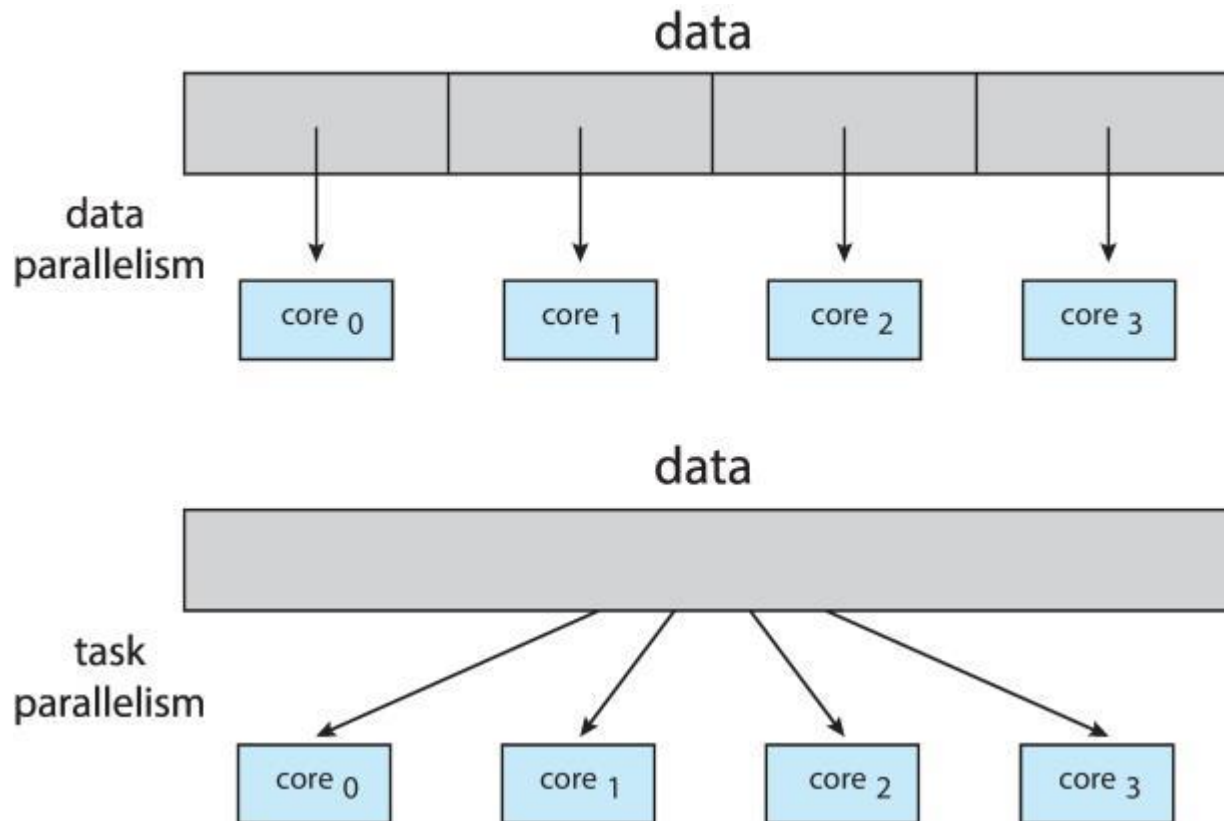
❑ Parallelism on a multi-core system:



Multicore Programming

- ❑ Types of parallelism
 - ❑ **Data parallelism** – **Data parallelism** focuses on **distributing subsets of the same data** across multiple computing cores and performing the same operation on each core.
 - ❑ **Task parallelism** – **Task parallelism** involves distributing not data but **tasks (threads)** across multiple computing cores.
- ❑ Example: lot of pixels of image or payroll cheques to update
 - ❑ DP: taking the data and dividing among multiple processors (add only bonus to all.. Divide 50-50 two processor)
 - ❑ TP: divide the tasks on 2 processor (avg, max, min salary), can use the same or different data

Data and Task Parallelism



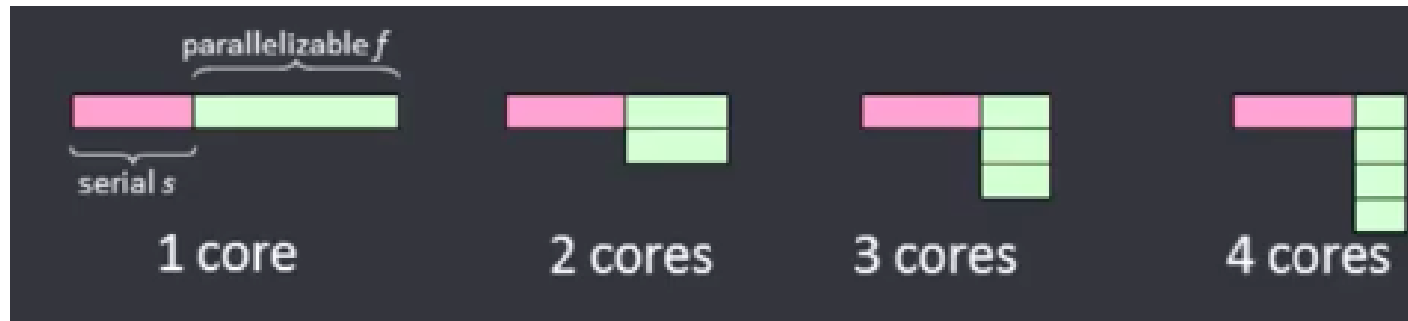
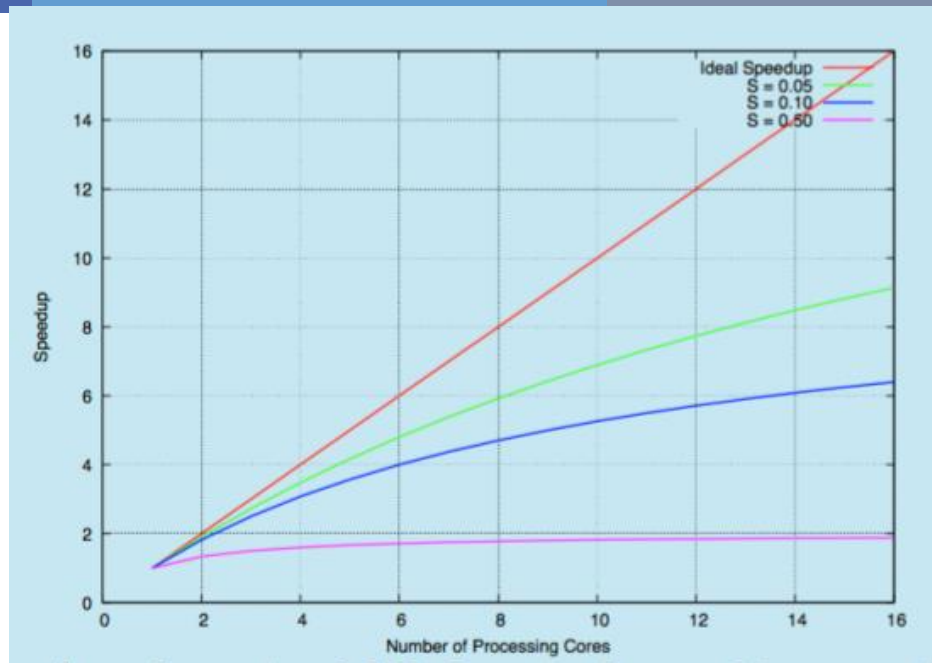
Amdahl's Law

- ❑ Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- ❑ S is serial portion
- ❑ $1-S$ parallel portion
- ❑ 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?

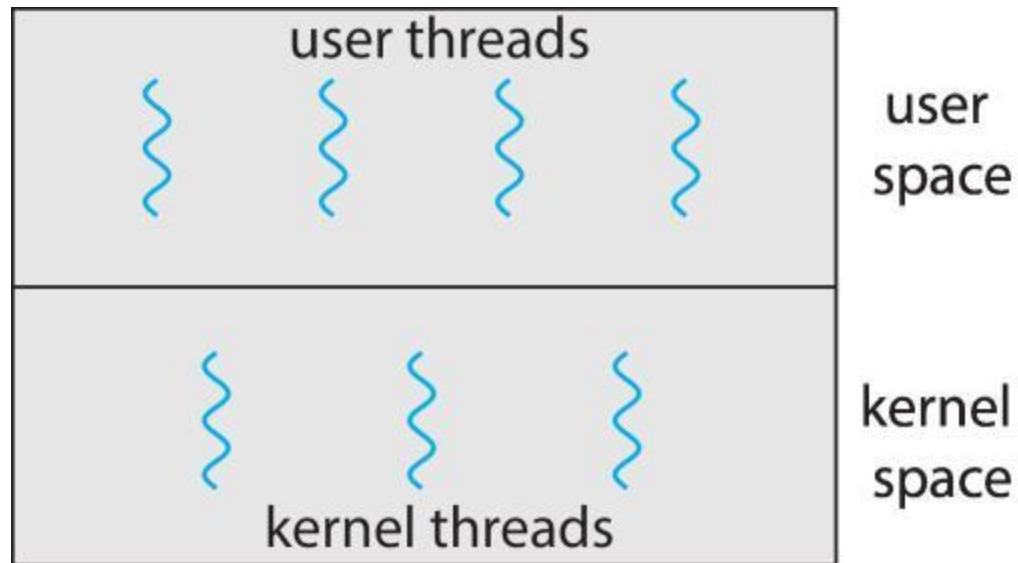
Amdahl's Law



User Threads and Kernel Threads

- ❑ **User threads** - management done by user-level threads library
- ❑ Three primary thread libraries:
 - ❑ POSIX **Pthreads**
 - ❑ Windows threads
 - ❑ Java threads
- ❑ **Kernel threads** - Supported by the Kernel
- ❑ Examples – virtually all general purpose operating systems, including:
 - ❑ Windows
 - ❑ Linux
 - ❑ Mac OS X
 - ❑ iOS
 - ❑ Android

User and Kernel Threads

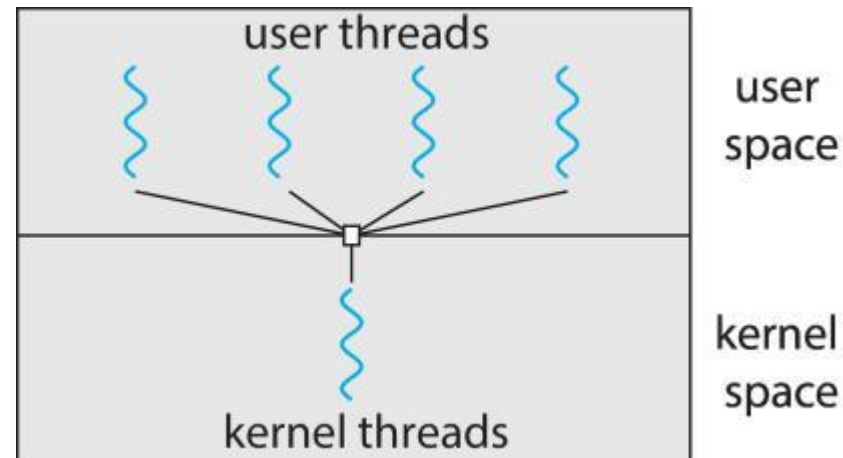


Multithreading Models

- ☐ 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 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**

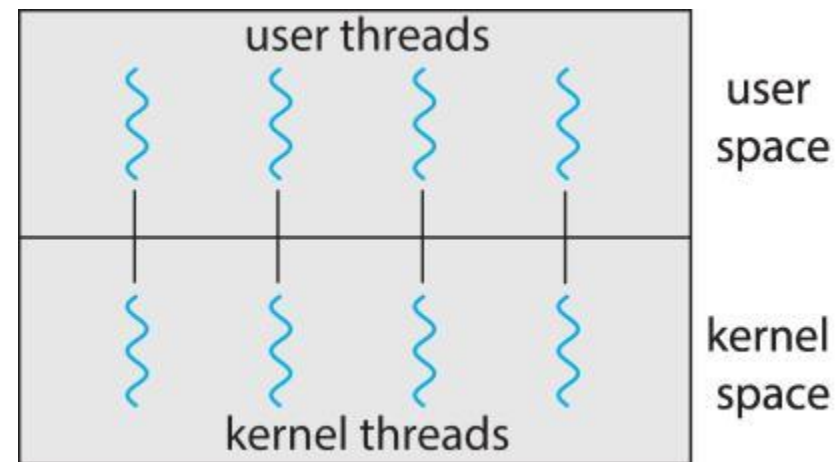


One-to-One

- ❑ Each user-level thread maps to 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

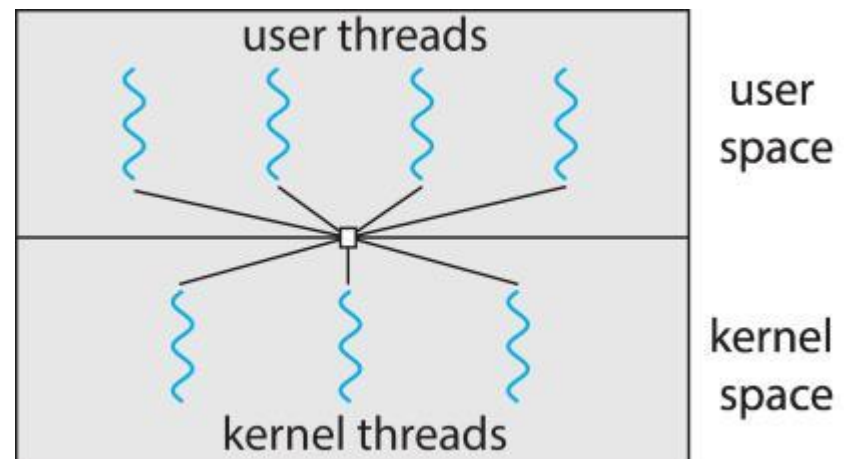
- ❑ Examples

- ❑ Windows
- ❑ Linux



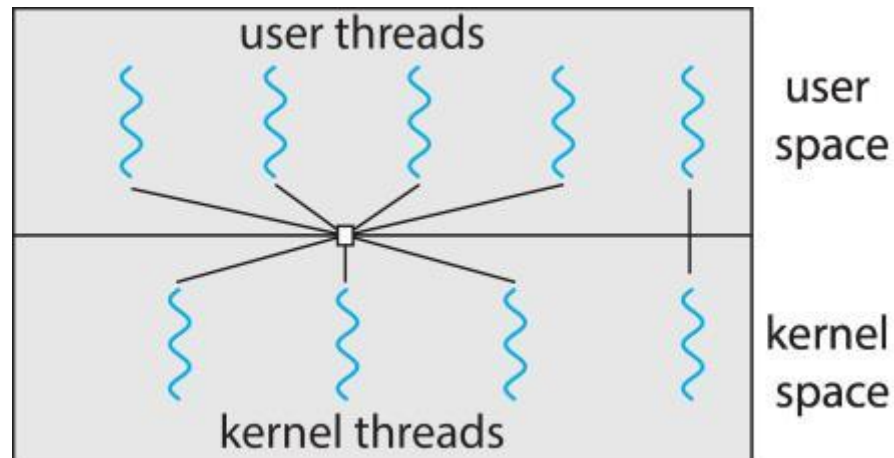
Many-to-Many Model

- ❑ Allows many user level threads to be mapped to many kernel threads
- ❑ Allows the operating system to create a sufficient number of kernel threads
- ❑ Windows with the *ThreadFiber* package
- ❑ Otherwise not very common



Two-level Model

- ❑ Similar to M:M, except that it allows a user thread to be **bound** to kernel thread



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

Pthreads

- ❑ **Pthreads** refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization
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 (Linux & Mac OS X)

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'd)

```
/* 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);
```

Windows Multithreaded C Program

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

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 1; i <= Upper; i++)
        Sum += i;
    return 0;
}
```


Windows Multithreaded C Program (cont'd)

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

    Param = atoi(argv[1]);
    /* 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 */

    /* 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
- ❑ Java threads may be created by:
 - ❑ Extending Thread class
 - ❑ Implementing the Runnable interface

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

- ❑ Standard practice is to implement Runnable interface

Java Threads

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) { }
```

Java Executor Framework

- ❑ Rather than explicitly creating threads, Java also allows thread creation around the Executor interface:

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

- ❑ The Executor is used as follows:

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

Java Executor Framework

```
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'd)

```
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) { }
    }
}
```

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

- ❑ 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
 - ❑ i.e.Tasks could be scheduled to run periodically
- ❑ Windows API supports thread pools:

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


Java Thread Pools

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

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

Java Thread Pools (cont'd)

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

public class ThreadPoolExample
{
    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0].trim());

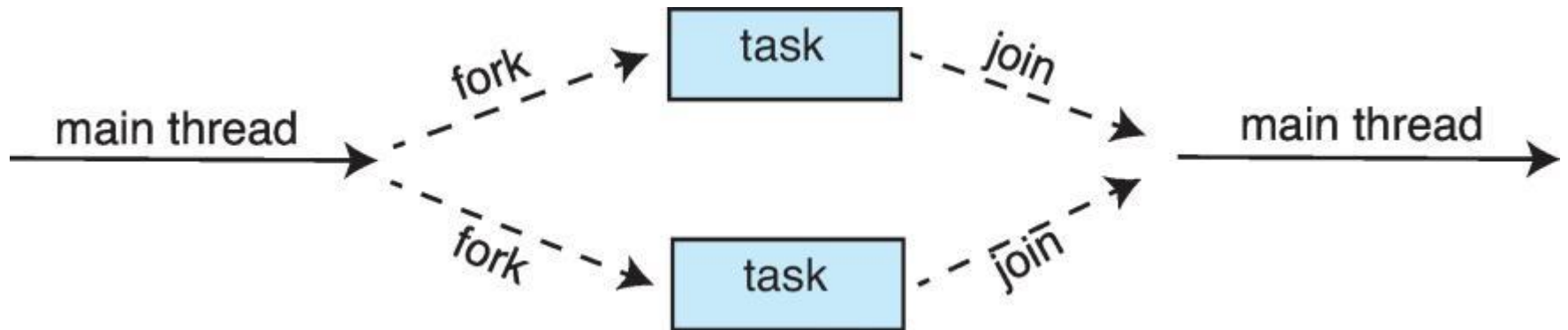
        /* Create the thread pool */
        ExecutorService pool = Executors.newCachedThreadPool();

        /* 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

- ❑ Multiple threads (tasks) are **forked**, and then **joined**.



Fork-Join Parallelism

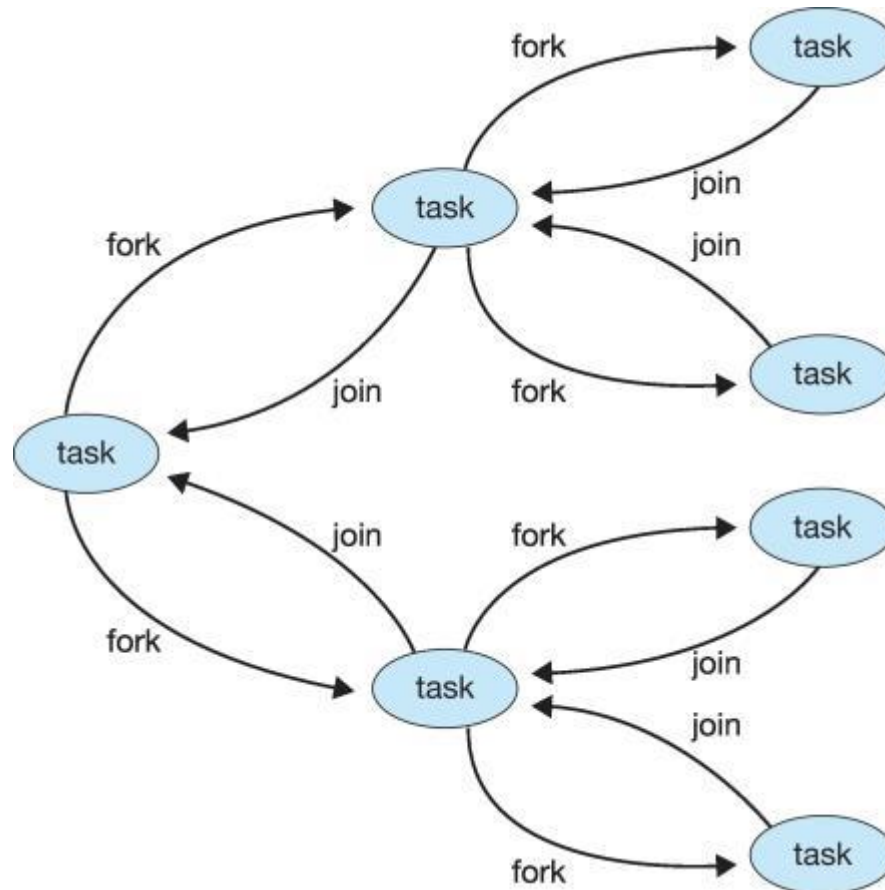
- General algorithm for fork-join strategy:

```
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



Fork-Join Parallelism in Java

```
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 Parallelism in 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;

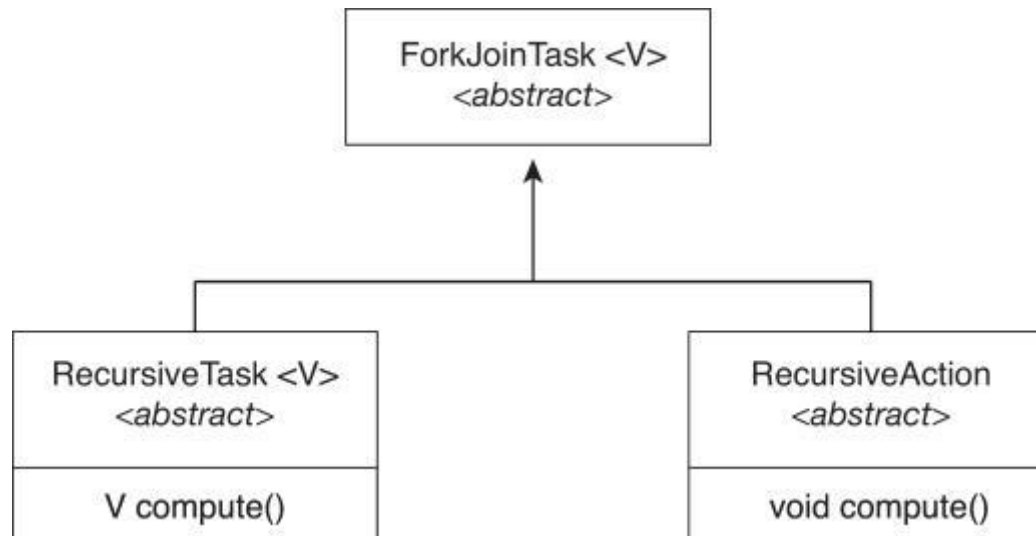
            SumTask leftTask = new SumTask(begin, mid, array);
            SumTask rightTask = new SumTask(mid + 1, end, array);

            leftTask.fork();
            rightTask.fork();

            return rightTask.join() + leftTask.join();
        }
    }
}
```

Fork-Join Parallelism in Java

- ❑ The **ForkJoinTask** is an abstract base class
- ❑ **RecursiveTask** and **RecursiveAction** classes extend **ForkJoinTask**
- ❑ **RecursiveTask** returns a result (via the return value from the **compute()** method)
- ❑ **RecursiveAction** does not return a result



Grand Central Dispatch

- ❑ 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.") })
```

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
 - ❑ Why and when to use
 - ❑ Exec() immediately after Fork(), then duplication of all process is not needed
 - ❑ No Exec() after Fork(), then duplication of all process is needed
- ❑ **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
 - ❑ Signal is generated by particular event
 - ❑ Signal is delivered to a process
 - ❑ Signal is handled by one of two signal handlers:
 - ❑ default
 - ❑ 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'd)

- ❑ 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'd)

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

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

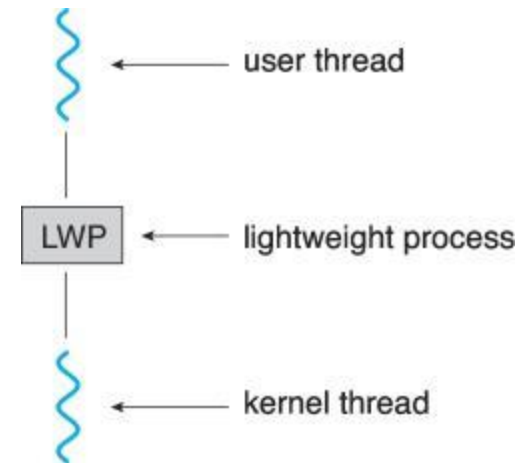
- ❑ If thread has cancellation disabled, cancellation remains pending until thread enables it
- ❑ Default type is deferred
 - ❑ Cancellation only occurs when thread reaches **cancellation point**
 - ❑ I.e. `pthread_testcancel()`
 - ❑ Then **cleanup handler** is invoked
- ❑ On Linux systems, thread cancellation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

Thread-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 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



Operating System Examples

- ☐ Windows Threads
- ☐ Linux Threads

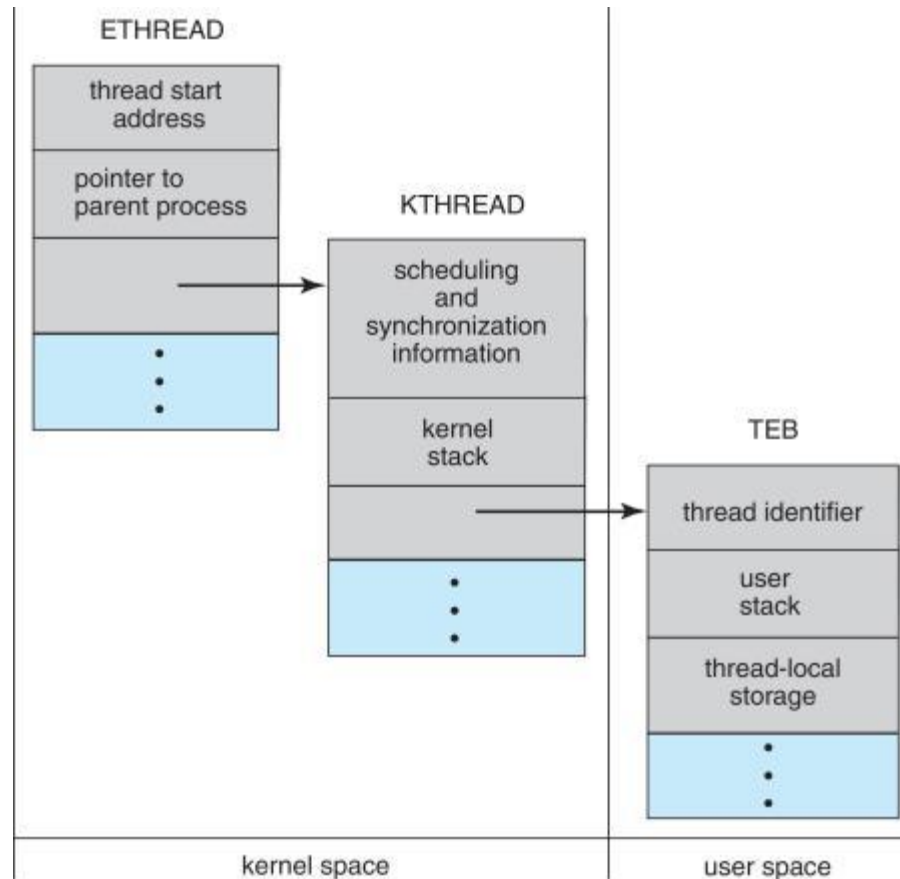
Windows Threads

- ❑ Windows API – primary API for Windows applications
- ❑ 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'd)

- ❑ 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 refers to them as **tasks** rather than **threads**
- ❑ Thread creation is done through **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)



Books

- ❑ Operating Systems Concept
 - ❑ Written by Galvin and Silberschatz
 - ❑ Edition: 9th



References

- ❑ Operating Systems Concept
 - ❑ Written by Galvin and Silberschatz
 - ❑ Edition: 9th