NPTEL ONLINE CERTIFICATION COURSES

# Operating System Fundamentals

Santanu Chattopadhyay
Electronics and Electrical Communication Engg.

# Threads

CONCEPTS COVERED

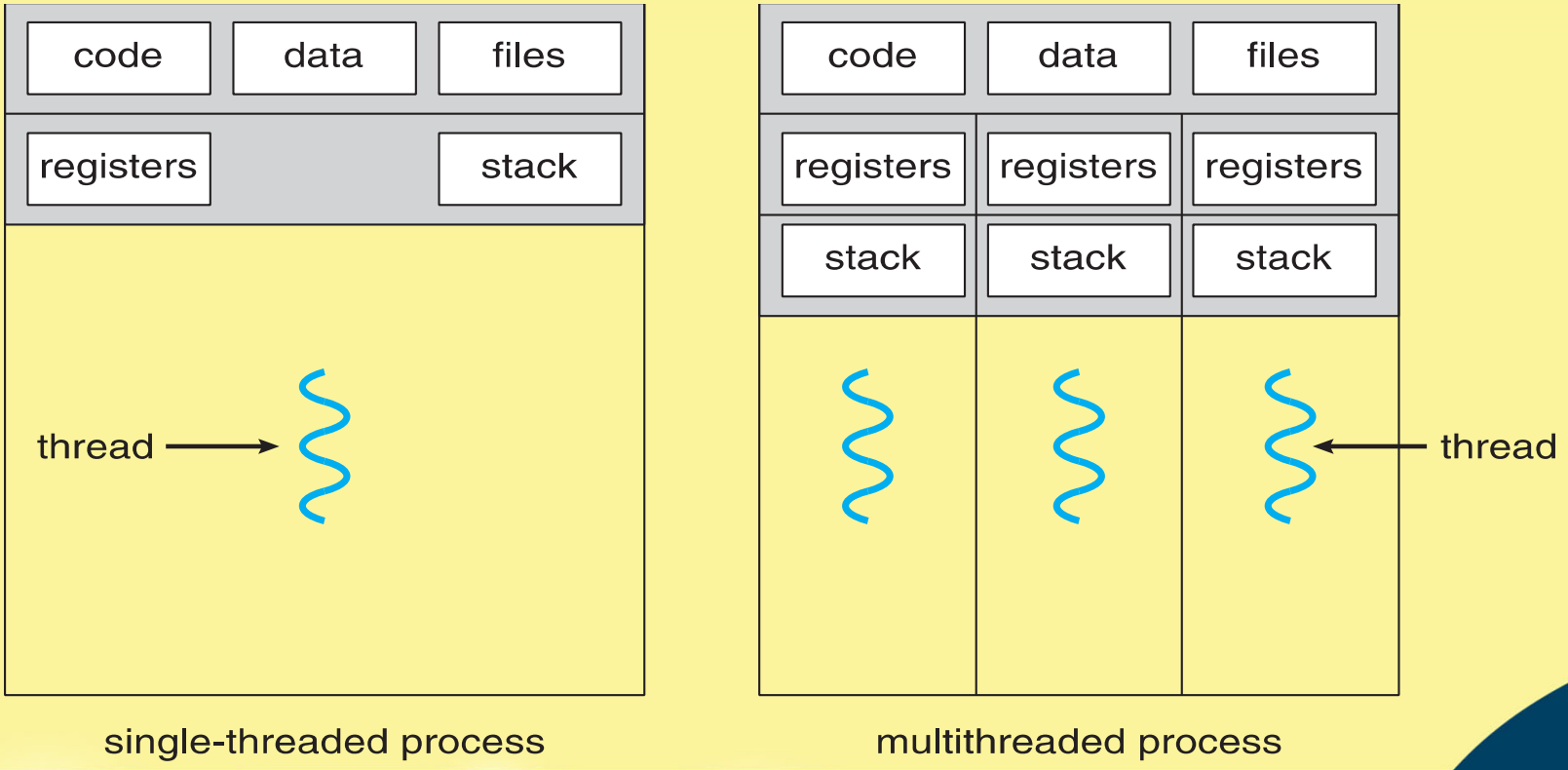
## Concepts Covered:

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


# 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

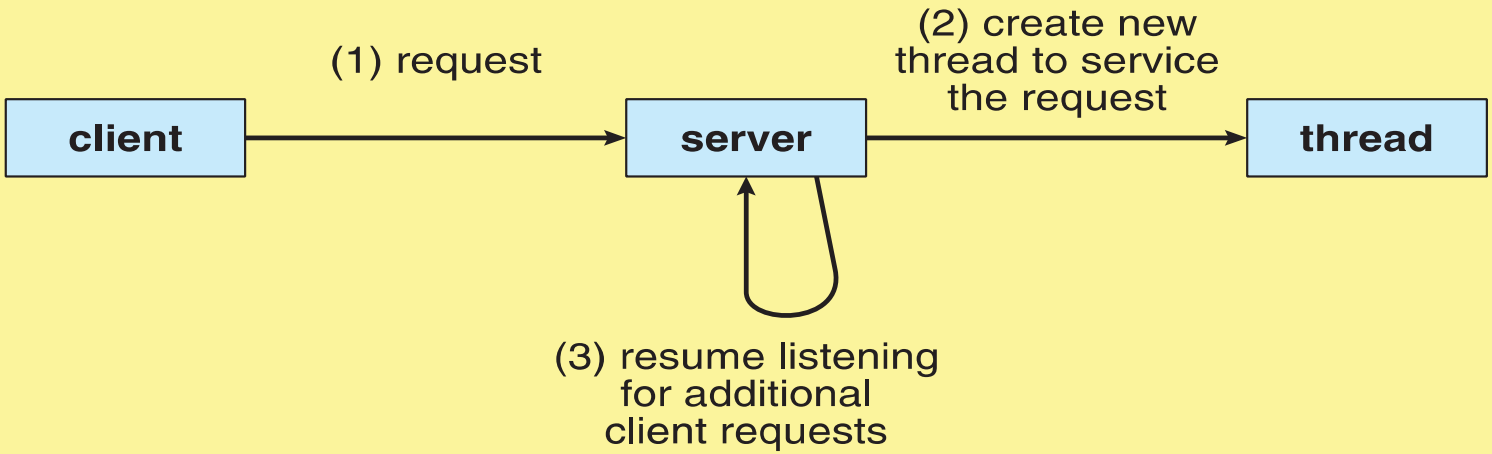

# Single and Multithreaded Processes




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 multiple cores, 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.)

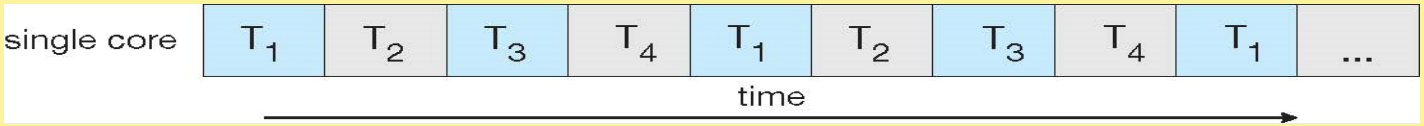
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As number of threads grows, so does architectural support for threading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

# Multicore Programming

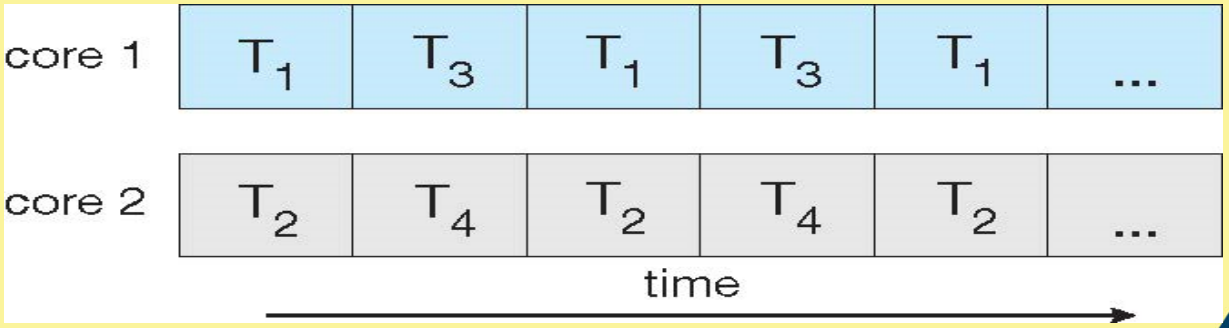
- **Multicore** or **multiprocessor** systems are placing pressure on programmers. Challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***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:**



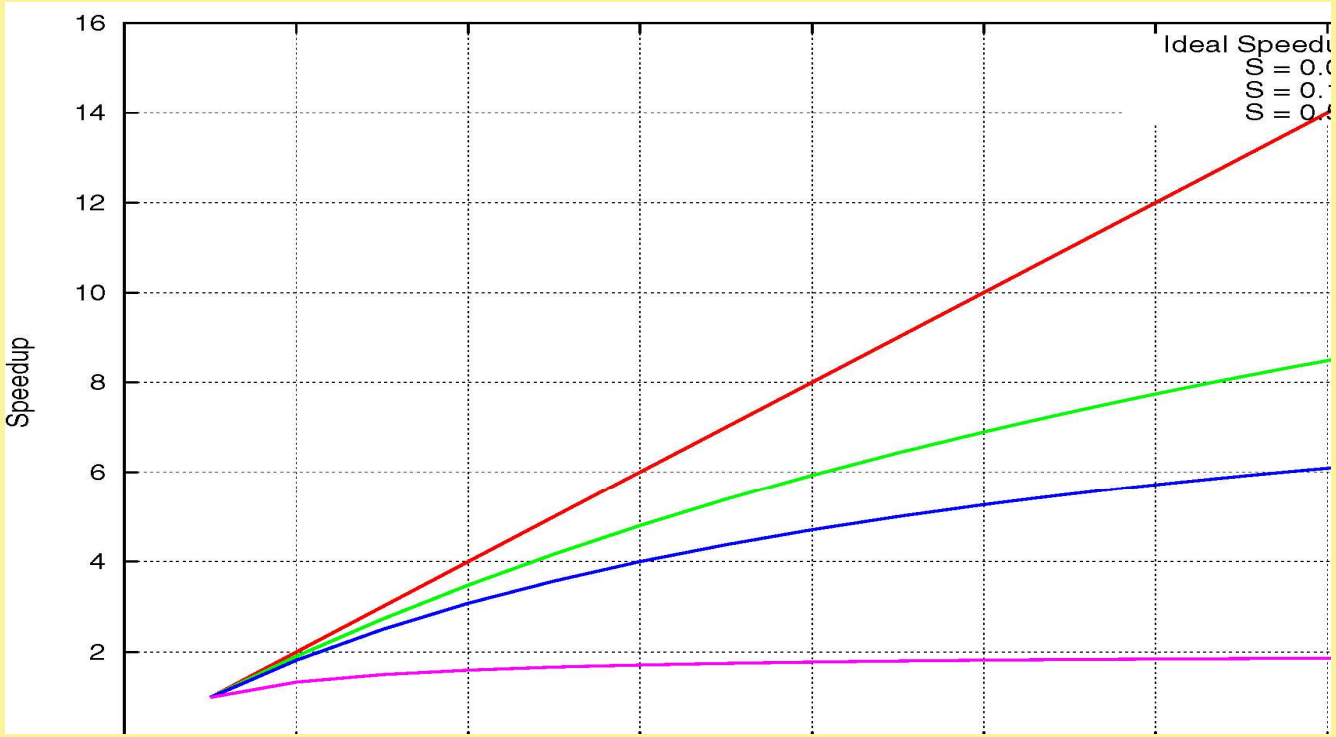
# Amdahl's Law

- 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?

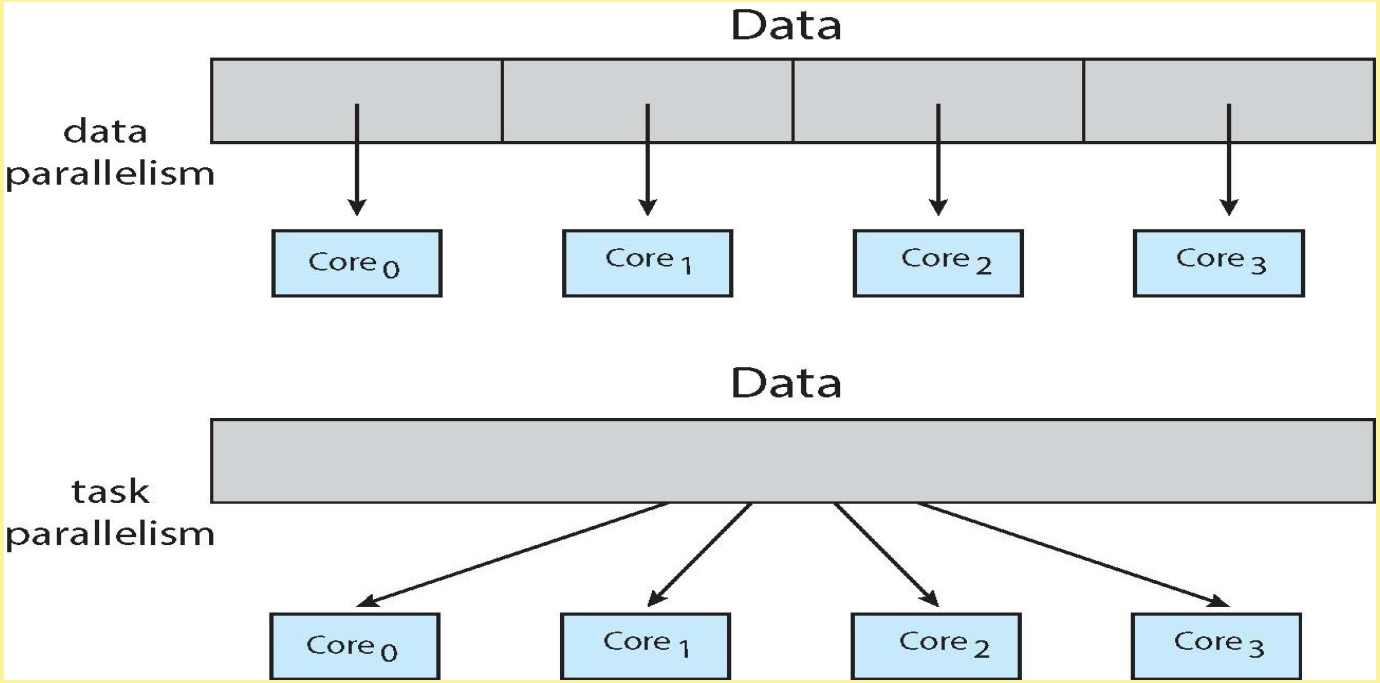

# Figure Amdahl



# 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.

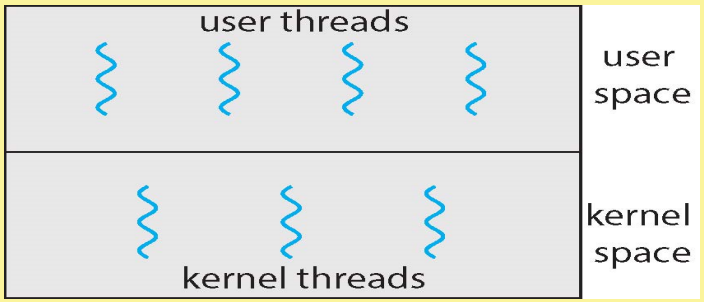
# Data and Task Parallelism



# 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 systems support kernel threads:
  - Windows, Linux, and Mac OS X

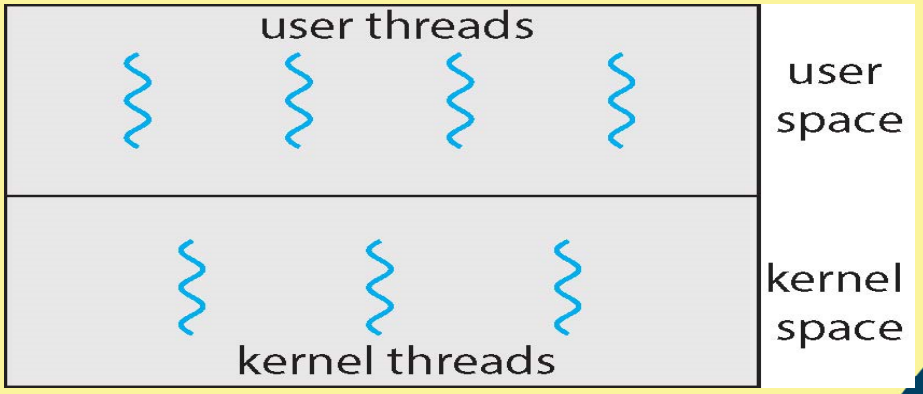


# Relationship between user and Kernel threads

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

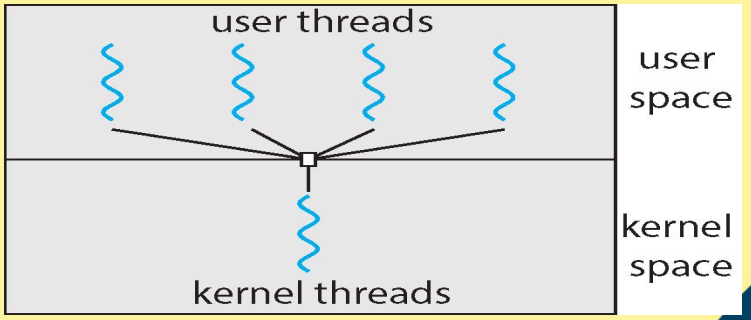
# One-to-One Model

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



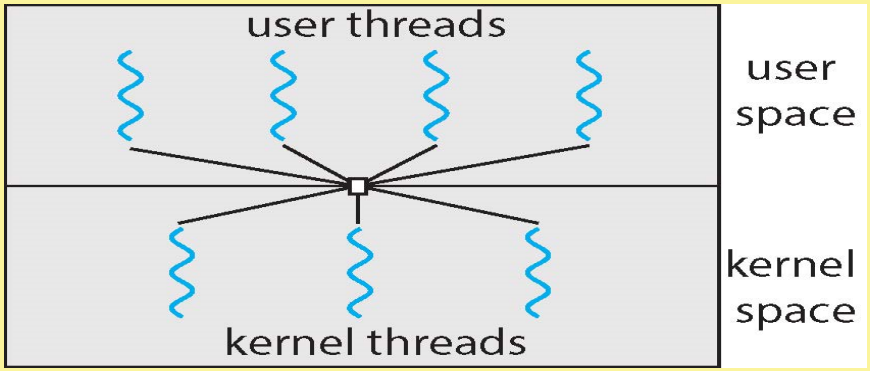
# Many-to-One Model

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



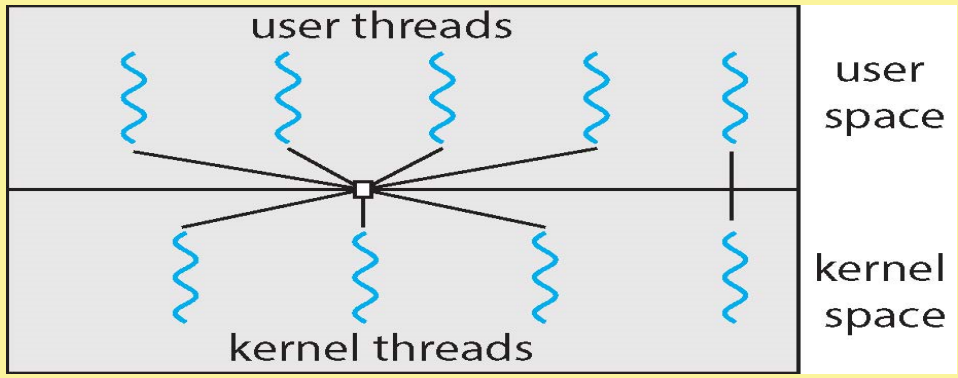
# Many-to-Many Model

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



# 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);
```


# 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 thread 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

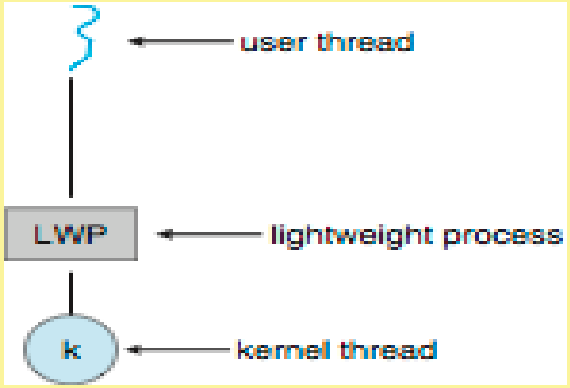
- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible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

Conclusion

## Conclusion:

- Thread is a flow of control within a process
- Benefits include increased responsiveness to the user, resource sharing, economy and ability to exploit multiprocessor architecture
- Threads may be at user-level or at kernel-level
- Models may be one-to-one, many-to-many

NPTEL ONLINE CERTIFICATION COURSES

Thank you

# Operating System Fundamentals

Santanu Chattopadhyay
Electronics and Electrical Communication Engg.

# CPU Scheduling

## Concepts Covered:

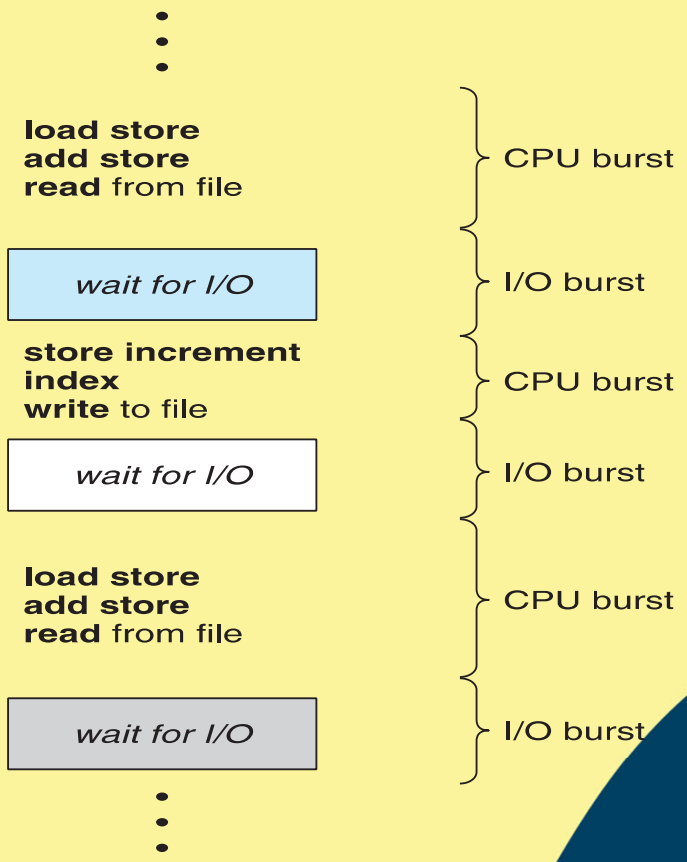
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling

# Objectives

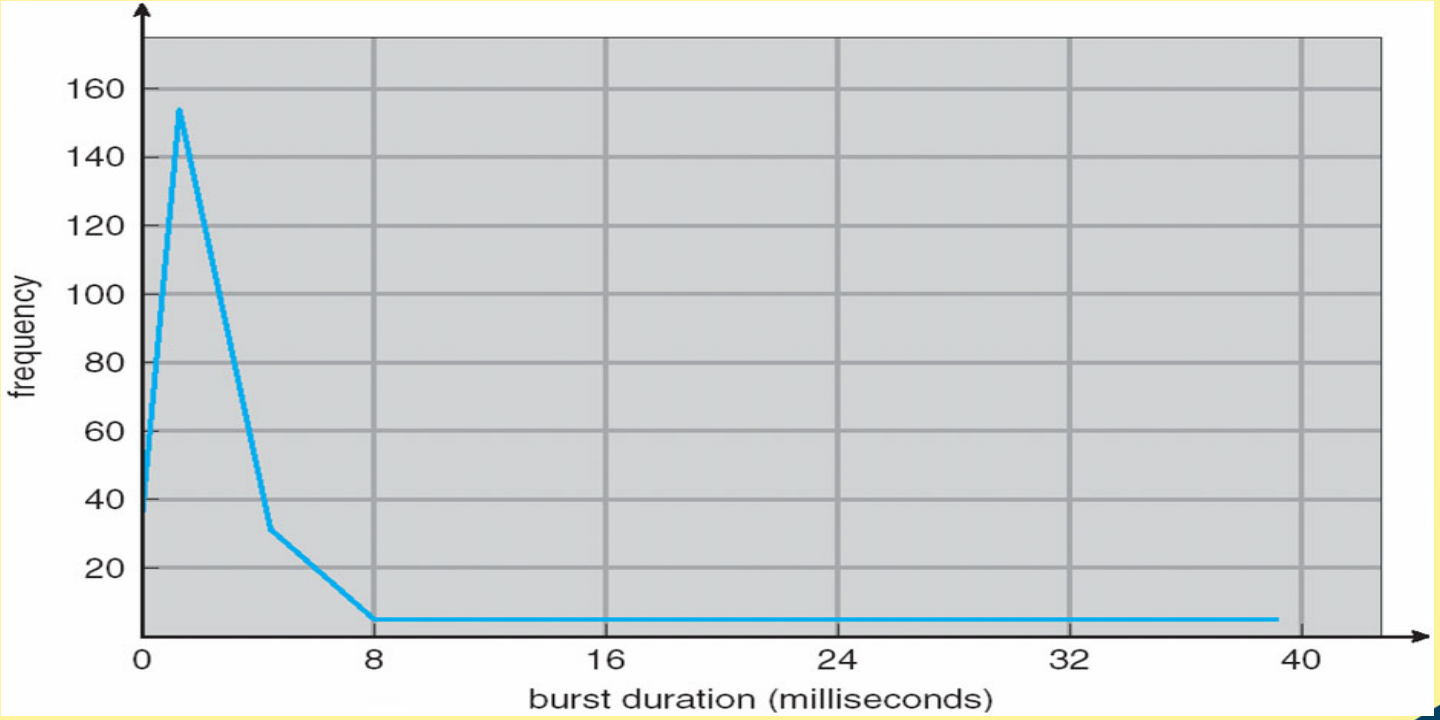
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems


# Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- Most processes exhibit the following behavior:
- **CPU burst** followed by **I/O burst**
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- CPU burst distribution is of main concern



# Histogram of CPU-burst Times



# CPU Scheduler

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the **CPU scheduler**.
- The ready queue may be ordered in various ways.
- CPU scheduling decisions may take place when a process:
  1. Switches from running state to waiting state
  2. Switches from running state to ready state
  3. Switches from waiting state to ready state
  4. When a process terminates
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- There is a choice, however, for situations 2 and 3.


# Nonpreemptive Scheduling

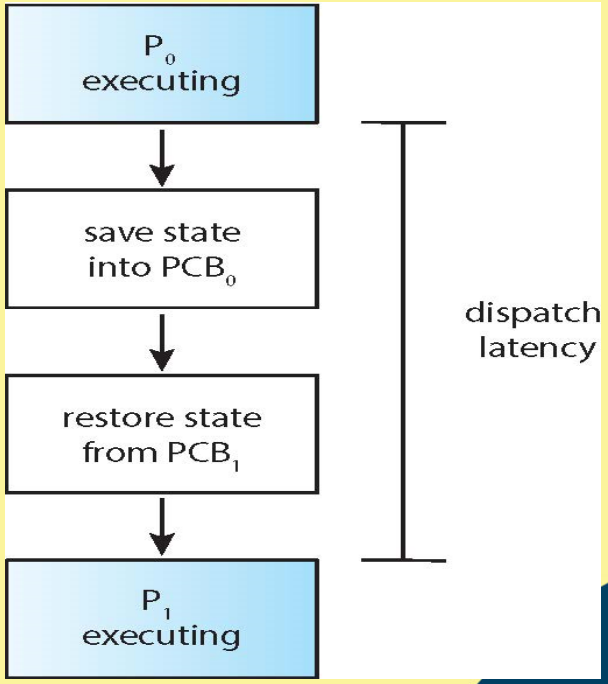
- Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU:
  - either by terminating
  - or by switching to the waiting state.

# Preemptive scheduling

- Preemptive scheduling can result in race conditions when data are shared among several processes.
  - Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities
- Virtually all modern operating systems including Windows, Mac OS X, Linux, and UNIX use preemptive scheduling algorithms.


# Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the CPU scheduler; this involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



# Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – number of processes that complete their execution per time unit (e.g., 5 per second)
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – total amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)


# Optimization Criteria for Scheduling

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

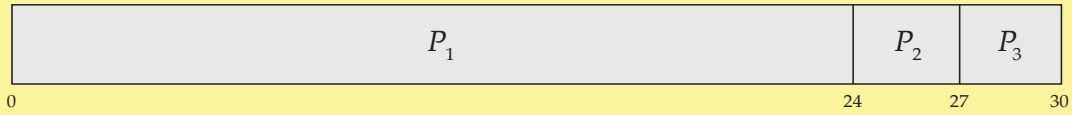
## Scheduling Algorithm

- First-come, First-serve (FCFS)
- Shortest-Job-First Scheduling (SJF)
- Round-Robin Scheduling (RR)
- Priority Scheduling
- Multilevel Queue Scheduling


# First- Come, First-Served (FCFS) Scheduling

- Consider the following three processes and their burst time

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |



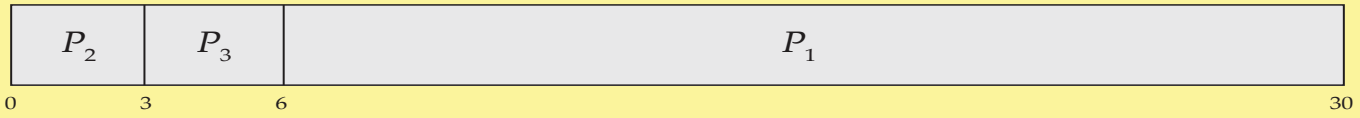

- Suppose that the processes arrive in the order: $P_1$ , $P_2$ , $P_3$
- We use Gantt Chart to illustrate a particular schedule
- Waiting time for $P_1$ = 0; $P_2$ = 24; $P_3$ = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

# FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:




- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

# Shortest-Job-First (SJF)

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - How do we know what is the length of the next CPU request
  - Could ask the user
    - What if the user lies?