

# Chapter 3: Processes

---





# Chapter 3: Processes

---

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems





# Objectives

---

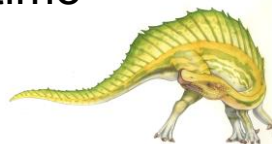
- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems





# Process Concept

- ❑ An operating system executes a variety of programs:
  - ❑ Batch system – **jobs**
  - ❑ Time-shared systems – **user programs** or **tasks**
- ❑ Textbook uses the terms **job** and **process** almost interchangeably
- ❑ **Process** – a program in execution; process execution must progress in sequential fashion
- ❑ Multiple parts
  - ❑ The program code, also called **text section**
  - ❑ Current activity including **program counter**, processor registers
  - ❑ **Stack** containing temporary data
    - ▶ Function parameters, return addresses, local variables
  - ❑ **Data section** containing global variables
  - ❑ **Heap** containing memory dynamically allocated during run time





# Process Concept (Cont.)

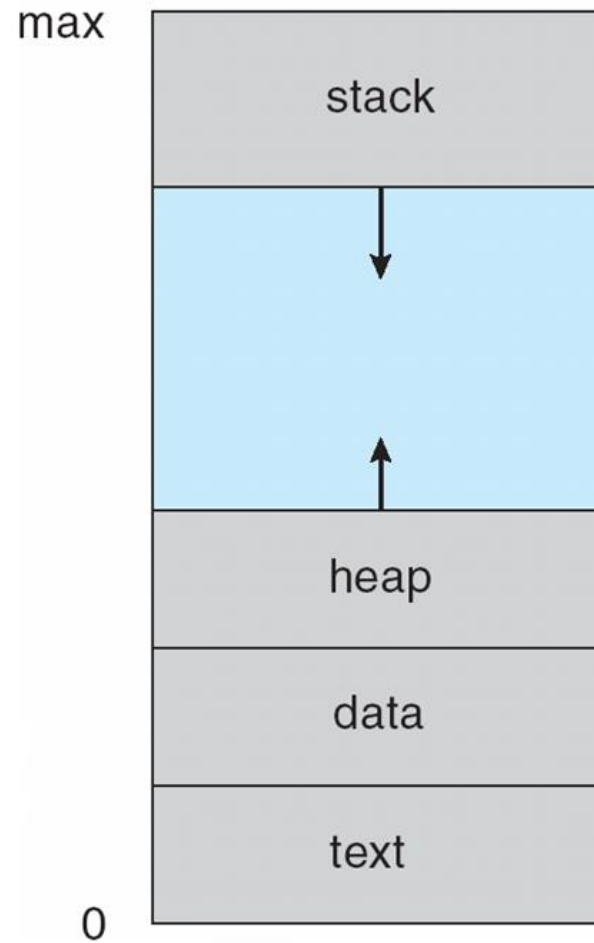
---

- Program is **passive** entity stored on disk (**executable file**), process is **active**
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program





# Process in Memory





# Process State

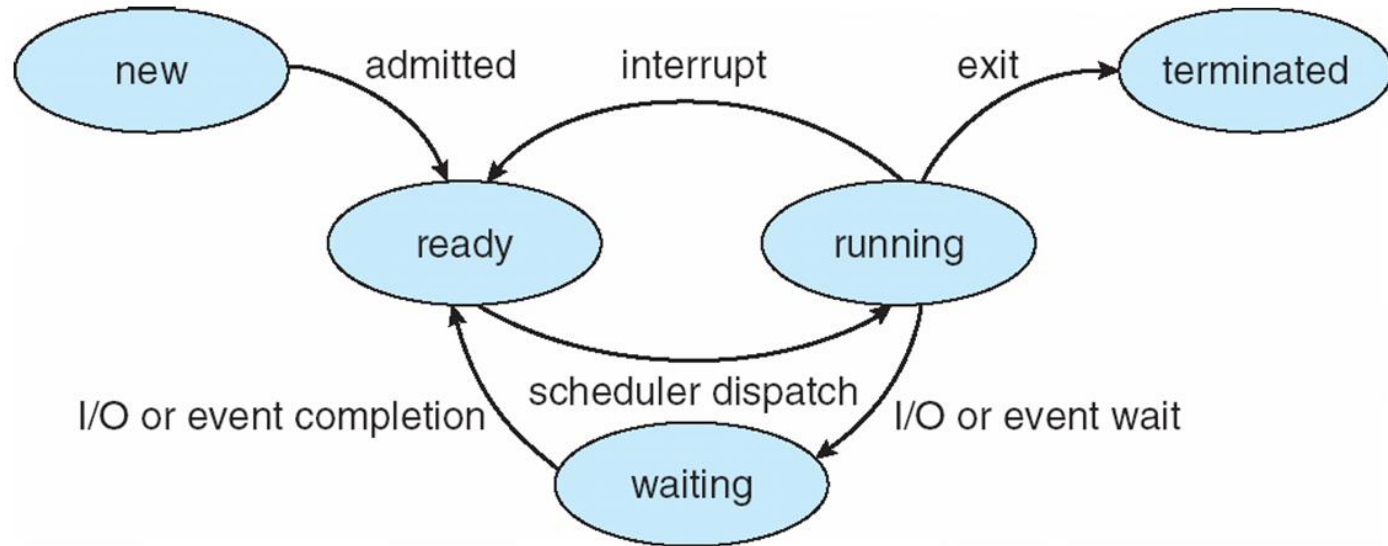
---

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution





# Diagram of Process State



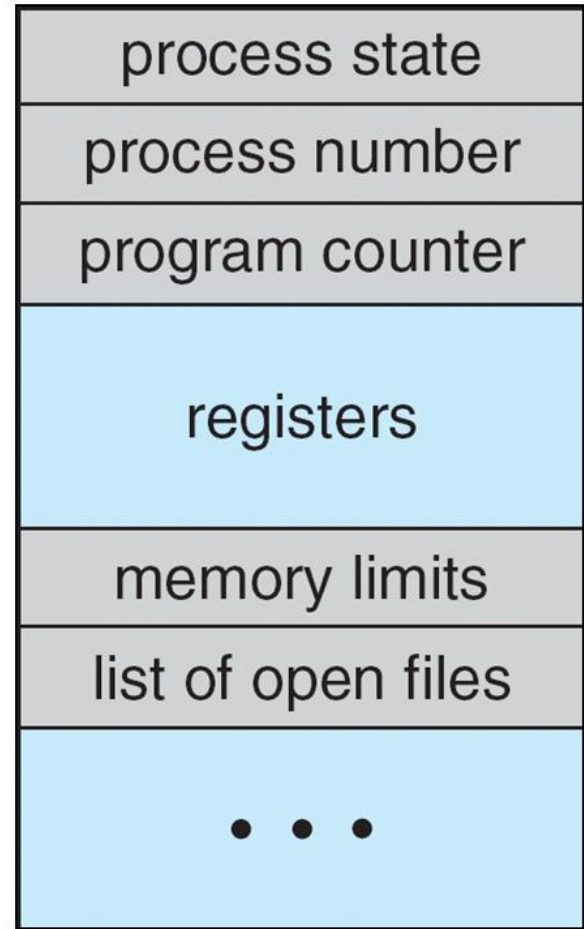




# Process Control Block (PCB)

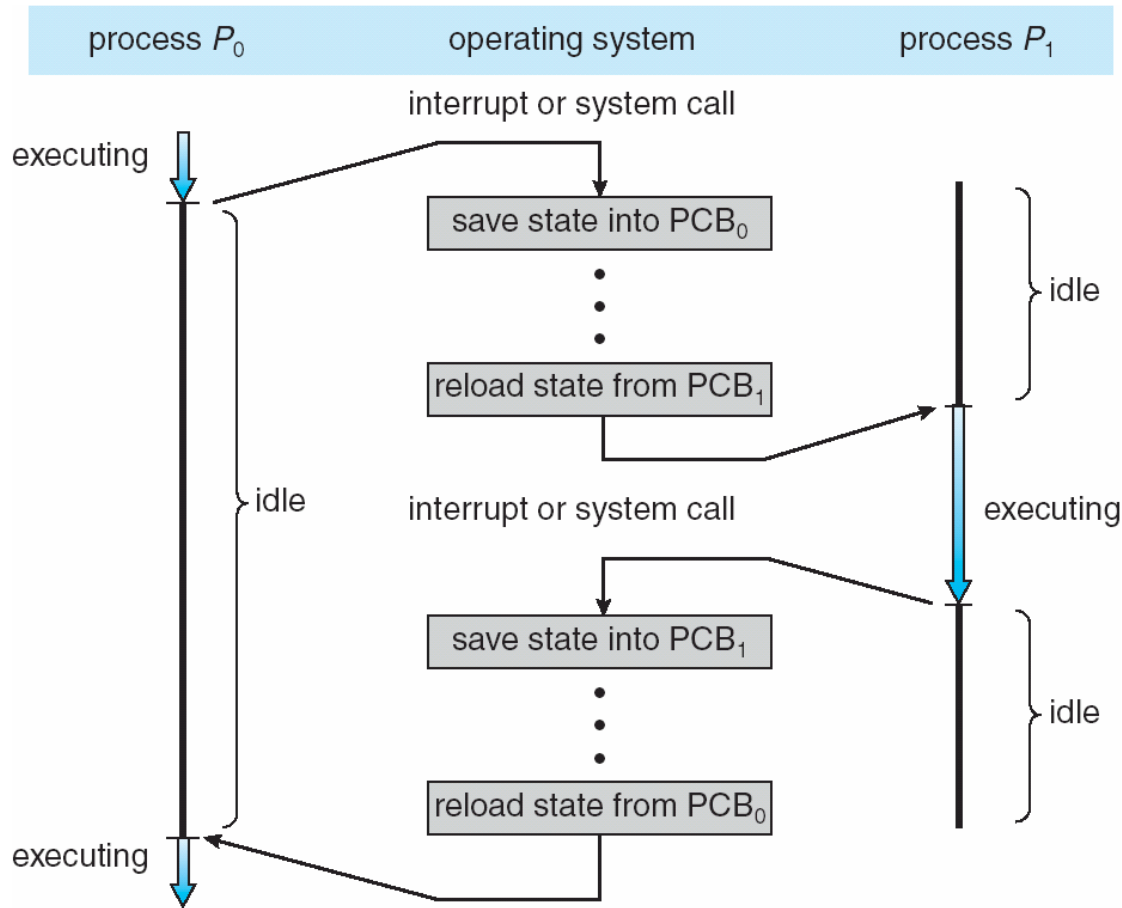
Information associated with each process  
(also called **task control block**)

- ❑ Process state – running, waiting, etc
- ❑ Program counter – location of instruction to next execute
- ❑ CPU registers – contents of all process-centric registers
- ❑ CPU scheduling information- priorities, scheduling queue pointers
- ❑ Memory-management information – memory allocated to the process
- ❑ Accounting information – CPU used, clock time elapsed since start, time limits
- ❑ I/O status information – I/O devices allocated to process, list of open files





# CPU Switch From Process to Process





# Threads

---

- ❑ So far, process has a single thread of execution
- ❑ Consider having multiple program counters per process
  - ❑ Multiple locations can execute at once
    - ▶ Multiple threads of control -> **threads**
- ❑ Must then have storage for thread details, multiple program counters in PCB
- ❑ See next chapter

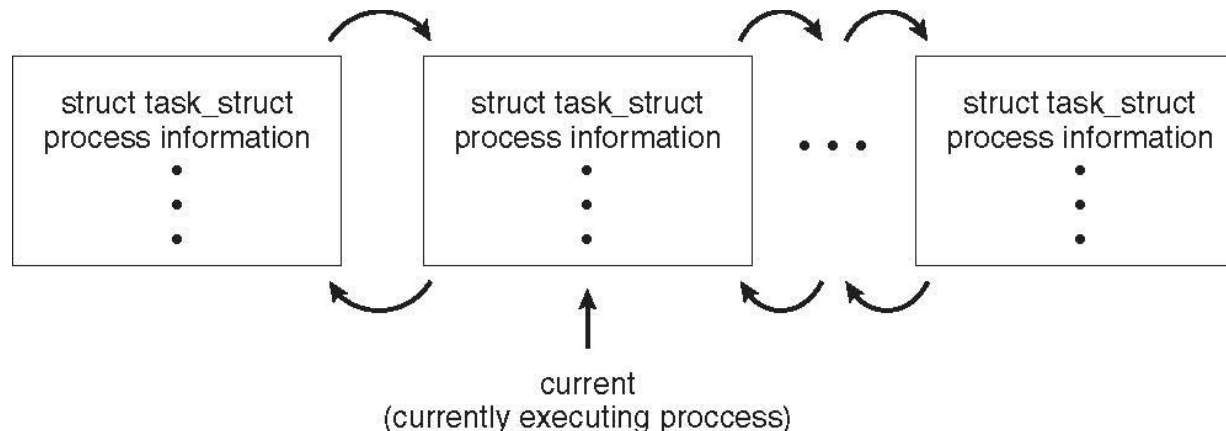




# Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```





# Process Scheduling

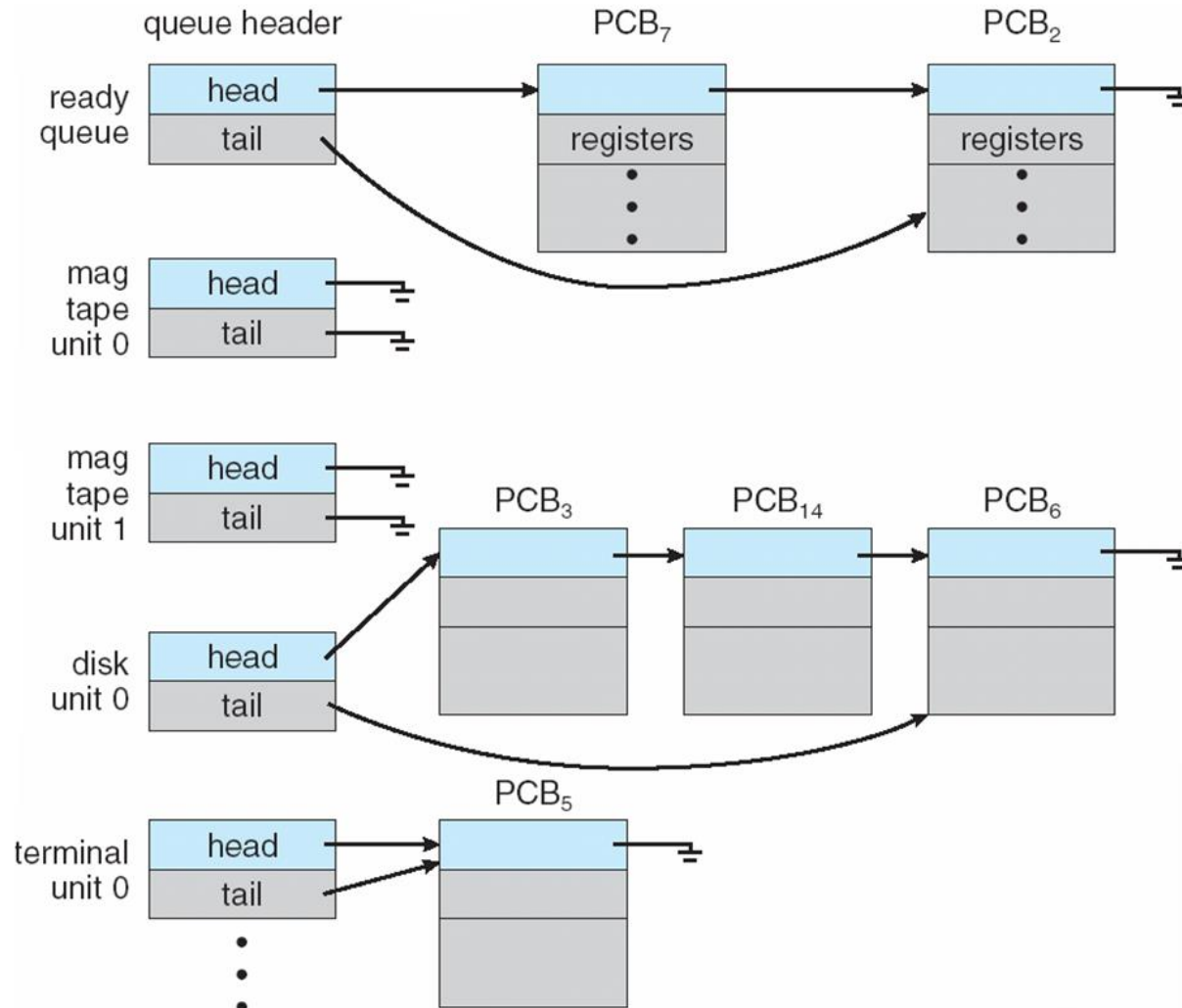
---

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues





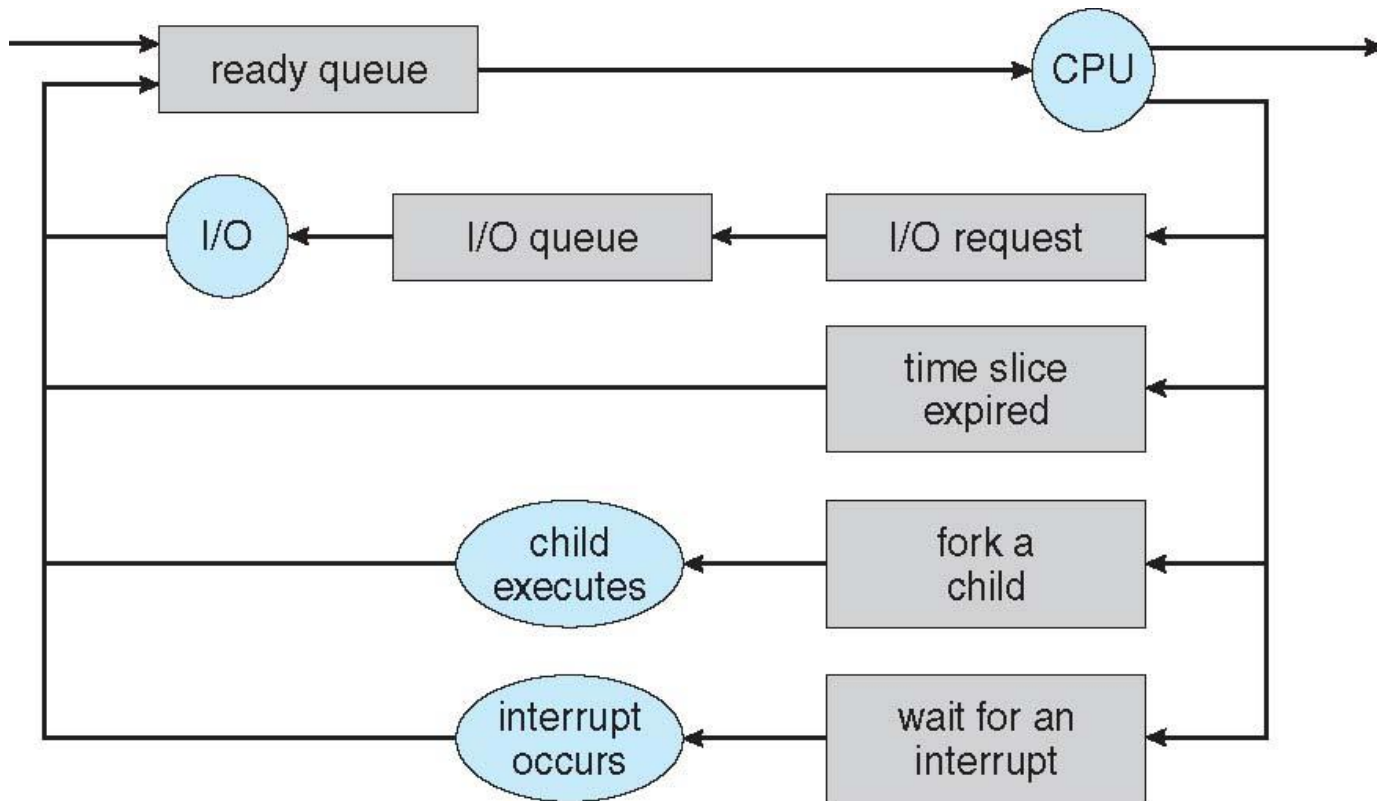
# Ready Queue And Various I/O Device Queues





# Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows





# Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)  $\Rightarrow$  (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes)  $\Rightarrow$  (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

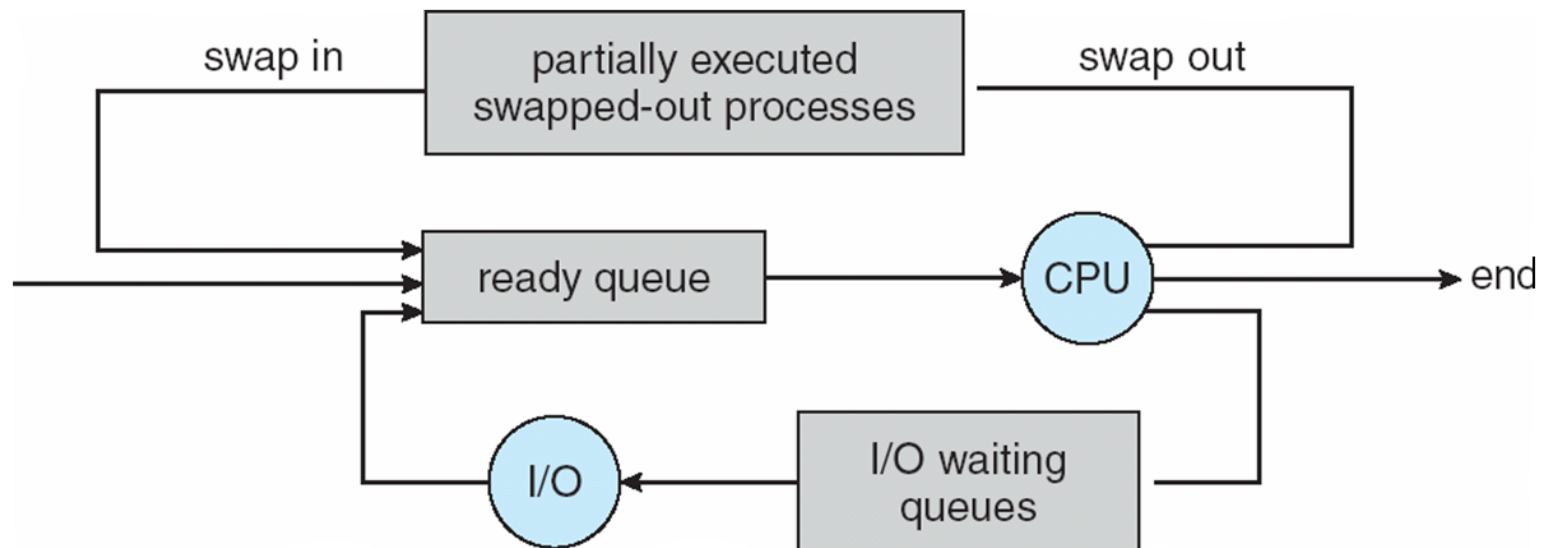






# Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**





# Multitasking in Mobile Systems

- ❑ Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- ❑ Due to screen real estate, user interface limits iOS provides for a
  - ❑ Single **foreground** process- controlled via user interface
  - ❑ Multiple **background** processes– in memory, running, but not on the display, and with limits
  - ❑ Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- ❑ Android runs foreground and background, with fewer limits
  - ❑ Background process uses a **service** to perform tasks
  - ❑ Service can keep running even if background process is suspended
  - ❑ Service has no user interface, small memory use





# Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once





# Operations on Processes

---

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next





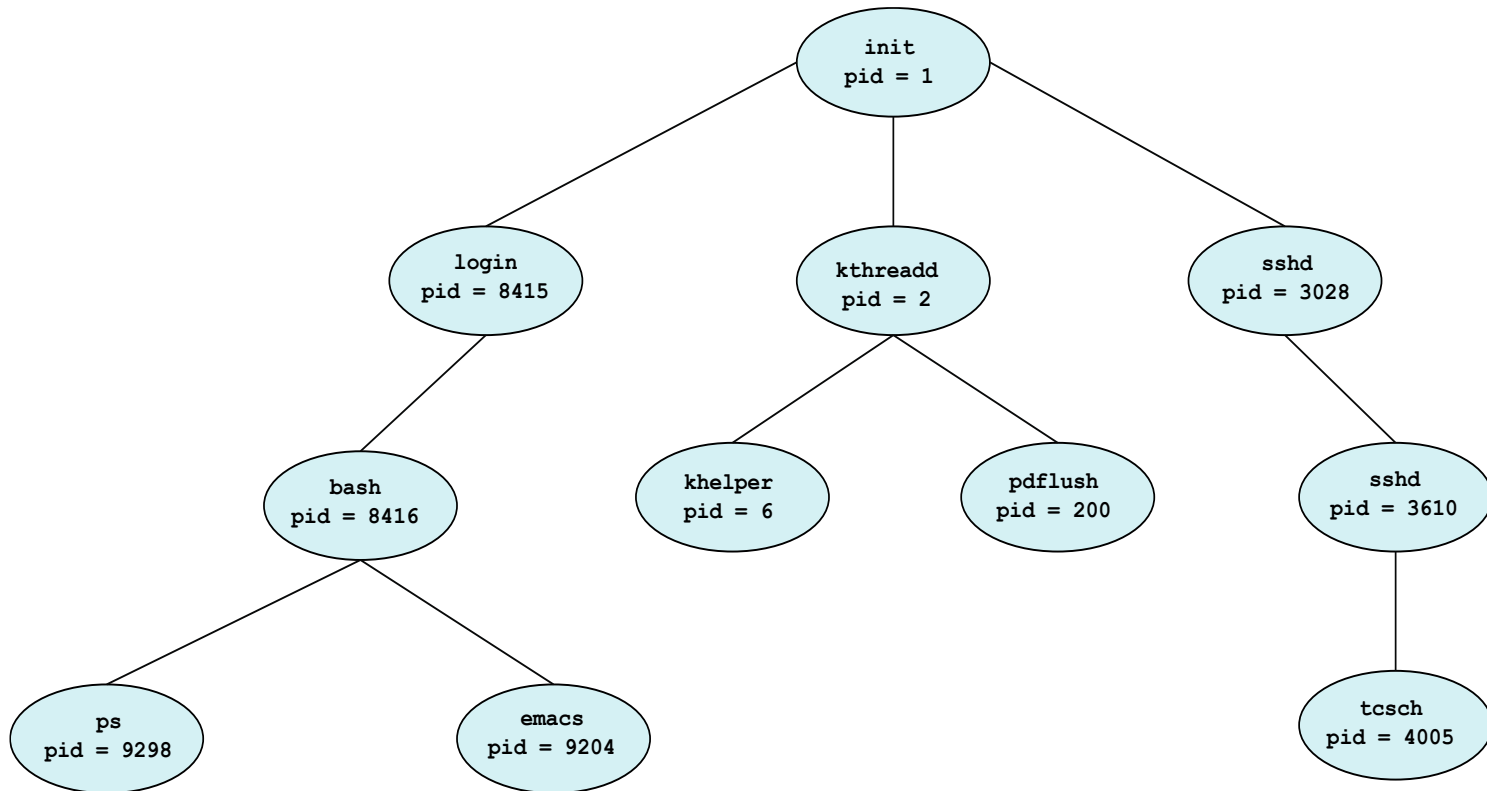
# Process Creation

- ❑ **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- ❑ Generally, process identified and managed via a **process identifier (pid)**
- ❑ Resource sharing options
  - ❑ Parent and children share all resources
  - ❑ Children share subset of parent's resources
  - ❑ Parent and child share no resources
- ❑ Execution options
  - ❑ Parent and children execute concurrently
  - ❑ Parent waits until children terminate





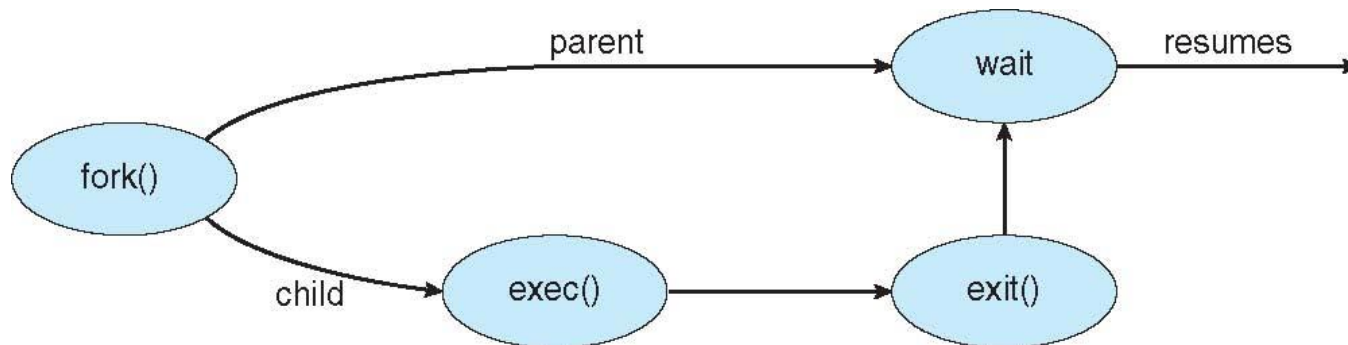
# A Tree of Processes in Linux





# Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork()** system call creates new process
  - **exec()** system call used after a **fork()** to replace the process' memory space with a new program





# C Program Forking Separate Process

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```







# Creating a Separate Process via Windows API

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

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
        "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
        NULL, /* don't inherit process handle */
        NULL, /* don't inherit thread handle */
        FALSE, /* disable handle inheritance */
        0, /* no creation flags */
        NULL, /* use parent's environment block */
        NULL, /* use parent's existing directory */
        &si,
        &pi))
    {
        fprintf(stderr, "Create Process Failed");
        return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```





# Process Termination

---

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
  - Returns status data from child to parent (via **wait()**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates





# Process Termination

- Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
    - **cascading termination.** All children, grandchildren, etc. are terminated.
    - The termination is initiated by the operating system.
  - The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process
- ```
pid = wait(&status);
```
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
  - If parent terminated without invoking `wait`, process is an **orphan**





# Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - ▶ Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in





# Interprocess Communication

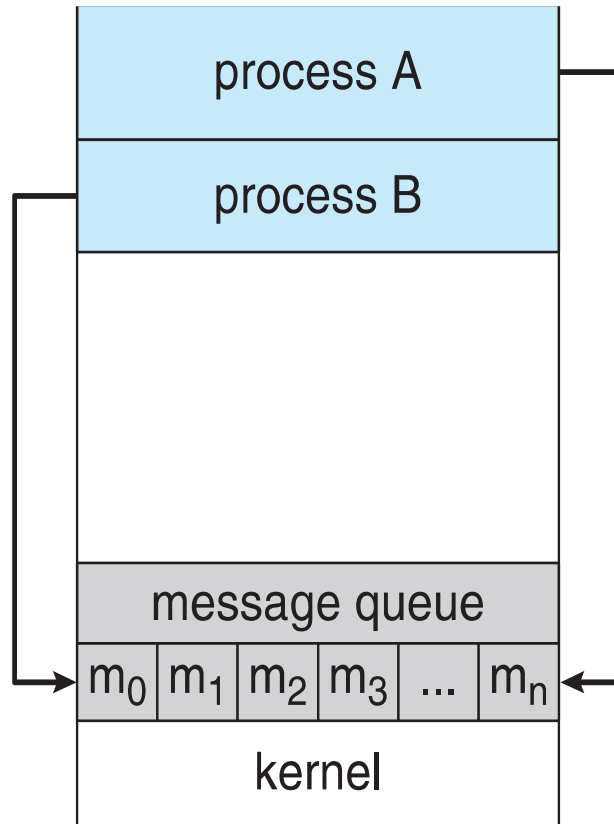
- ❑ Processes within a system may be *independent* or *cooperating*
- ❑ Cooperating process can affect or be affected by other processes, including sharing data
- ❑ Reasons for cooperating processes:
  - ❑ Information sharing
  - ❑ Computation speedup
  - ❑ Modularity
  - ❑ Convenience
- ❑ Cooperating processes need **interprocess communication (IPC)**
- ❑ Two models of IPC
  - ❑ **Shared memory**
  - ❑ **Message passing**



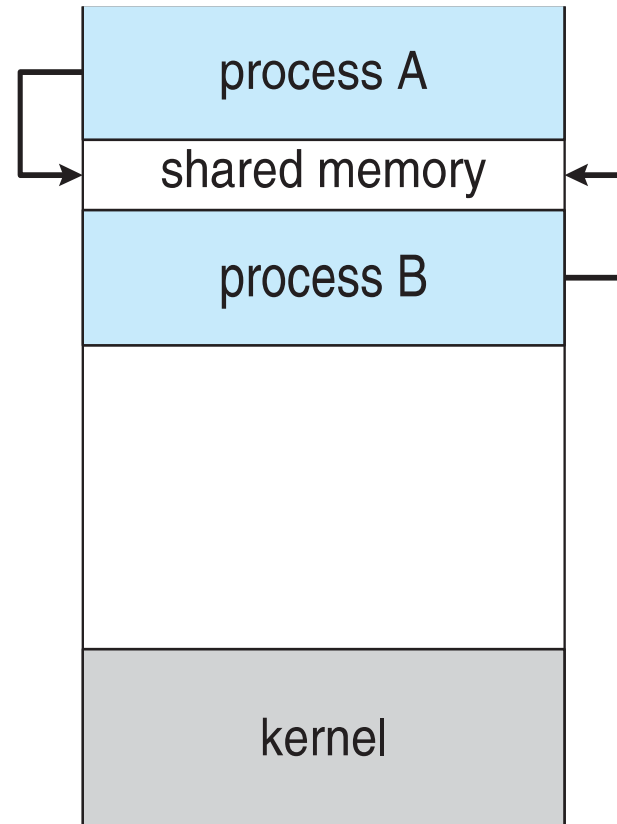


# Communications Models

(a) Message passing. (b) shared memory.



(a)



(b)





# Cooperating Processes

---

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience





# Producer-Consumer Problem

---

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size







# Bounded-Buffer – Shared-Memory Solution

## □ Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

## □ Solution is correct, but can only use BUFFER\_SIZE-1 elements





# Bounded-Buffer – Producer

---

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```





# Bounded Buffer – Consumer

---

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```





# Interprocess Communication – Shared Memory

---

- ❑ An area of memory shared among the processes that wish to communicate
- ❑ The communication is under the control of the users processes not the operating system.
- ❑ Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- ❑ Synchronization is discussed in great details in Chapter 5.





# Interprocess Communication – Message Passing

---

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable





## Message Passing (Cont.)

---

- If processes  $P$  and  $Q$  wish to communicate, they need to:
  - Establish a **communication link** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?





# Message Passing (Cont.)

---

- Implementation of communication link
  - Physical:
    - ▶ Shared memory
    - ▶ Hardware bus
    - ▶ Network
  - Logical:
    - ▶ Direct or indirect
    - ▶ Synchronous or asynchronous
    - ▶ Automatic or explicit buffering





# Direct Communication

---

- Processes must name each other explicitly:
  - **send** ( $P$ , *message*) – send a message to process  $P$
  - **receive**( $Q$ , *message*) – receive a message from process  $Q$
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional







# Indirect Communication

---

- ❑ Messages are directed and received from mailboxes (also referred to as ports)
  - ❑ Each mailbox has a unique id
  - ❑ Processes can communicate only if they share a mailbox
- ❑ Properties of communication link
  - ❑ Link established only if processes share a common mailbox
  - ❑ A link may be associated with many processes
  - ❑ Each pair of processes may share several communication links
  - ❑ Link may be unidirectional or bi-directional





# Indirect Communication

---

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send**(*A, message*) – send a message to mailbox *A*
  - receive**(*A, message*) – receive a message from mailbox *A*





# Indirect Communication

- Mailbox sharing
  - $P_1$ ,  $P_2$ , and  $P_3$  share mailbox A
  - $P_1$ , sends;  $P_2$  and  $P_3$  receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.





# Synchronization

- ? Message passing may be either blocking or non-blocking
- ? **Blocking** is considered **synchronous**
  - ? **Blocking send** -- the sender is blocked until the message is received
  - ? **Blocking receive** -- the receiver is blocked until a message is available
- ? **Non-blocking** is considered **asynchronous**
  - ? **Non-blocking send** -- the sender sends the message and continue
  - ? **Non-blocking receive** -- the receiver receives:
    - ? A valid message, or
    - ? Null message
- ? Different combinations possible
  - ? If both send and receive are blocking, we have a **rendezvous**





# Synchronization (Cont.)

---

## □ Producer-consumer becomes trivial

```
message next_produced;  
while (true) {  
    /* produce an item in next produced */  
    send(next_produced);  
}
```

```
message next_consumed;  
while (true) {  
    receive(next_consumed);  
  
    /* consume the item in next consumed */  
}
```





# Buffering

---

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.  
Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full
  3. Unbounded capacity – infinite length  
Sender never waits





# Examples of IPC Systems - POSIX

---

## ? POSIX Shared Memory

- ? Process first creates shared memory segment

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- ? Also used to open an existing segment to share it

- ? Set the size of the object

```
ftruncate(shm_fd, 4096);
```

- ? Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```





# IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```







# IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s", (char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```





# Examples of IPC Systems - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer  
`msg_send()`, `msg_receive()`, `msg_rpc()`
  - Mailboxes needed for communication, created via  
`port_allocate()`
  - Send and receive are flexible, for example four options if mailbox full:
    - ▶ Wait indefinitely
    - ▶ Wait at most n milliseconds
    - ▶ Return immediately
    - ▶ Temporarily cache a message





# Examples of IPC Systems – Windows

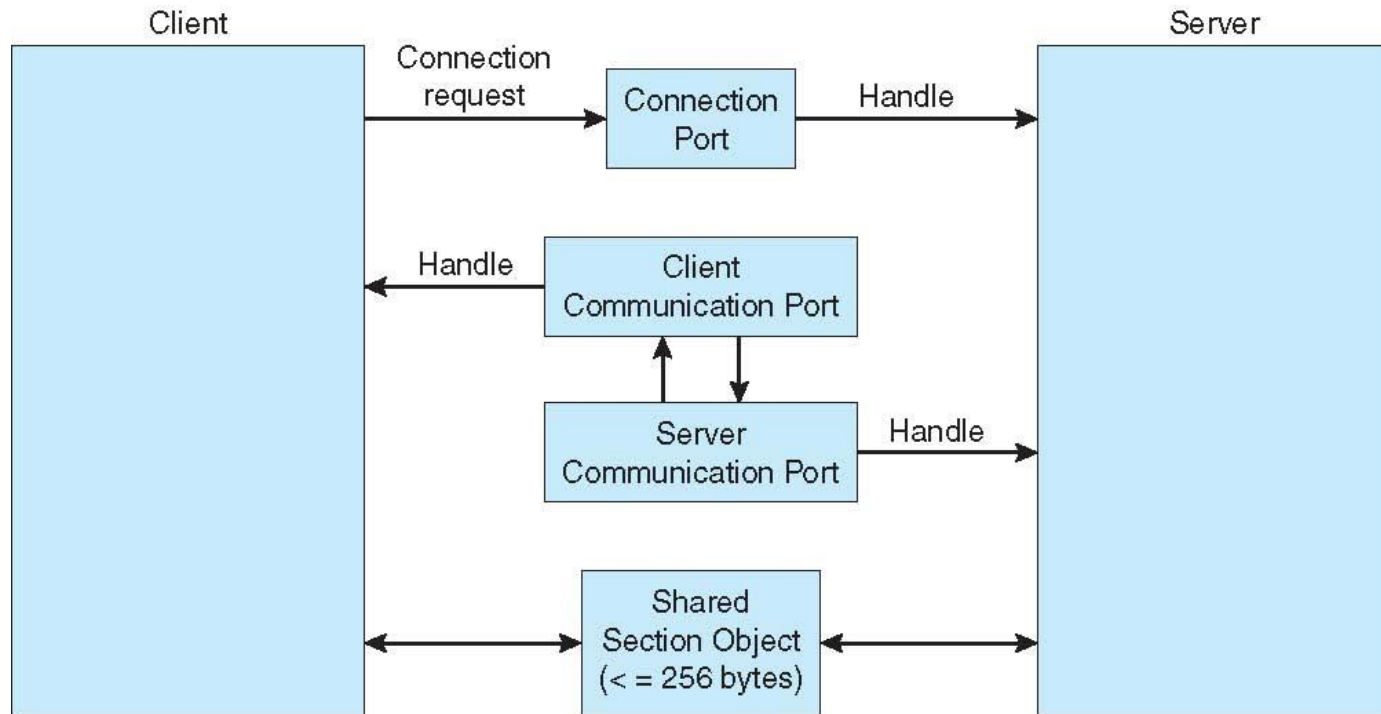
---

- Message-passing centric via **advanced local procedure call (LPC)** facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels
  - Communication works as follows:
    - ▶ The client opens a handle to the subsystem's **connection port** object.
    - ▶ The client sends a connection request.
    - ▶ The server creates two private **communication ports** and returns the handle to one of them to the client.
    - ▶ The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.





# Local Procedure Calls in Windows





# Communications in Client-Server Systems

---

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)





# Sockets

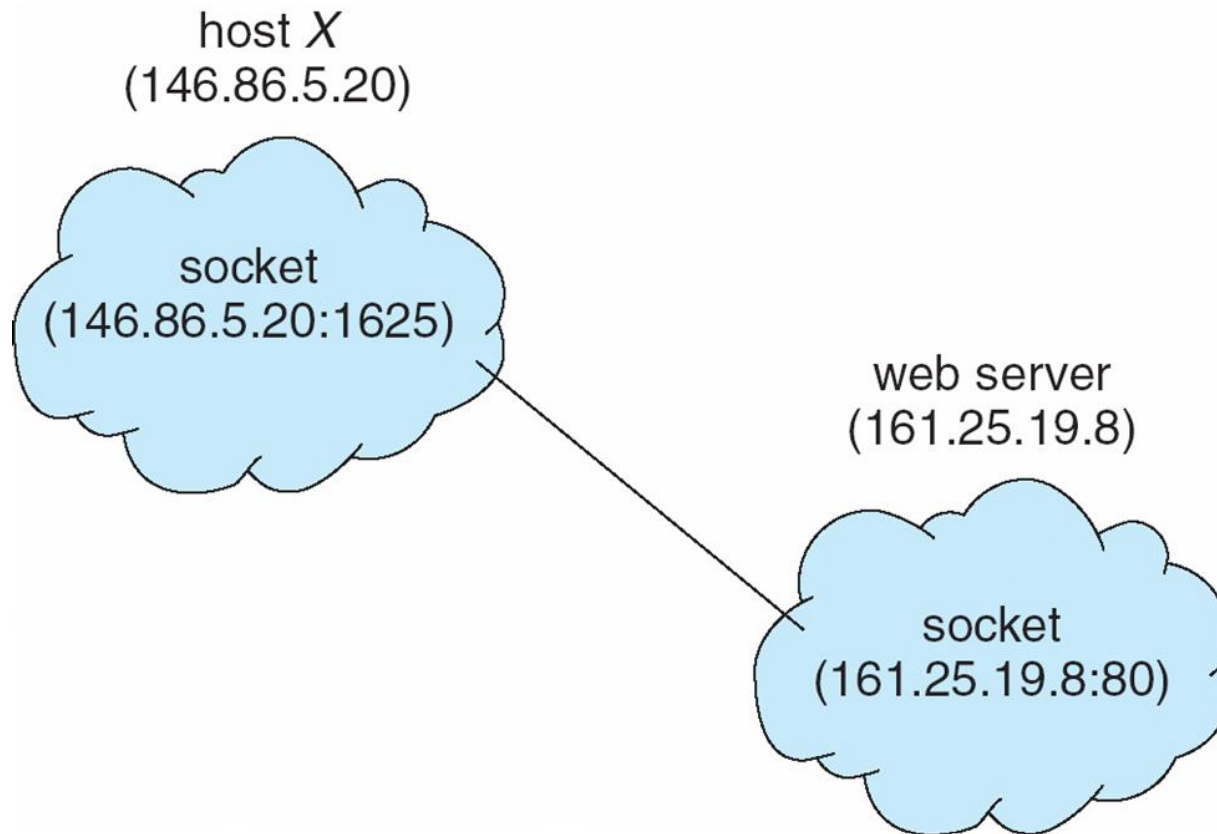
---

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are **well known**, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running





# Socket Communication





# Sockets in Java

- Three types of sockets
  - **Connection-oriented (TCP)**
  - **Connectionless (UDP)**
  - **MulticastSocket** class— data can be sent to multiple recipients
- Consider this “Date” server:

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            /* now listen for connections */
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                /* write the Date to the socket */
                pout.println(new java.util.Date().toString());

                /* close the socket and resume */
                /* listening for connections */
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```







# Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language (MIDL)**





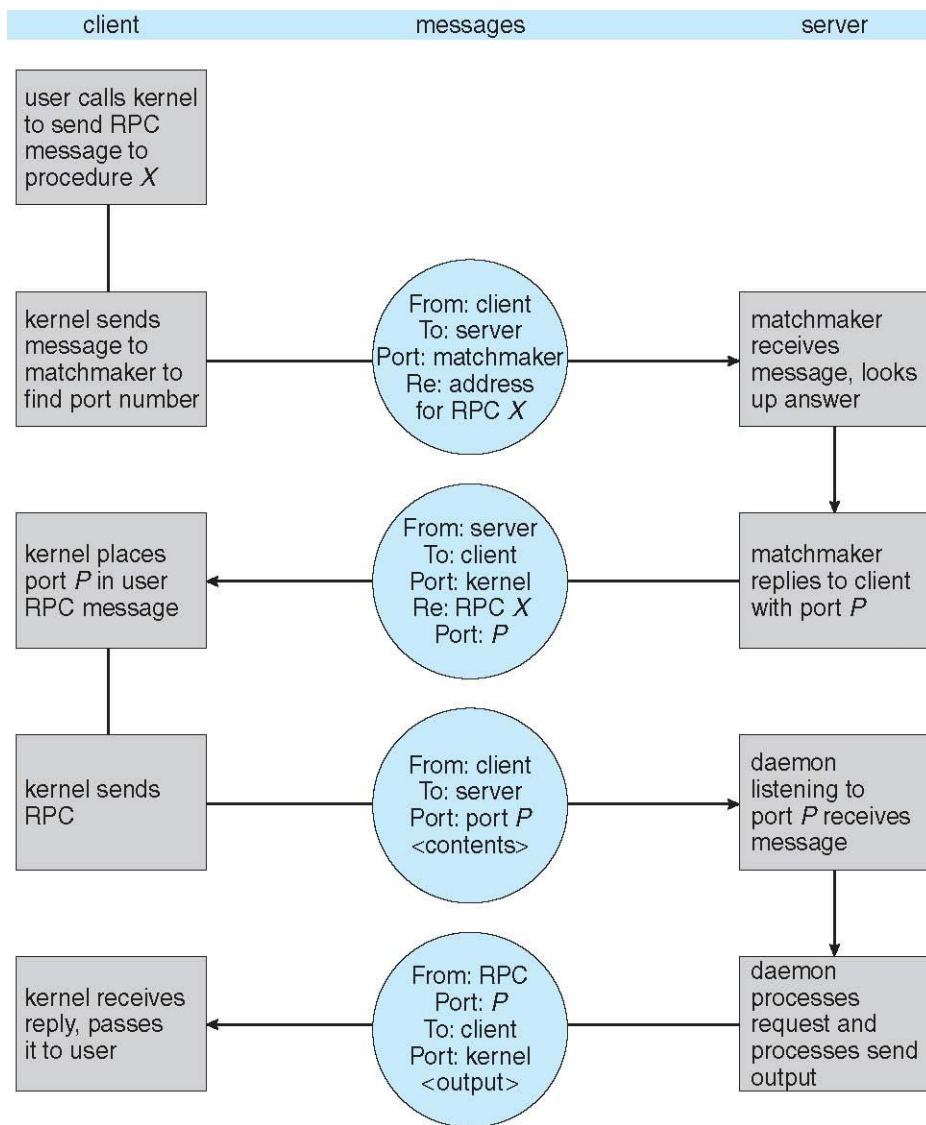
# Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation (XDL)** format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server





# Execution of RPC





# Pipes

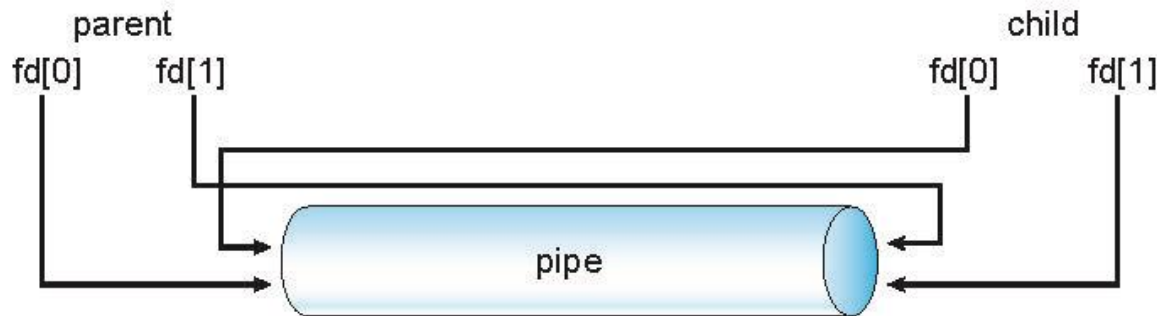
- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., **parent-child**) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.





# Ordinary Pipes

- ❑ Ordinary Pipes allow communication in standard producer-consumer style
- ❑ Producer writes to one end (the **write-end** of the pipe)
- ❑ Consumer reads from the other end (the **read-end** of the pipe)
- ❑ Ordinary pipes are therefore unidirectional
- ❑ Require parent-child relationship between communicating processes



- ❑ Windows calls these **anonymous pipes**
- ❑ See Unix and Windows code samples in textbook





# Named Pipes

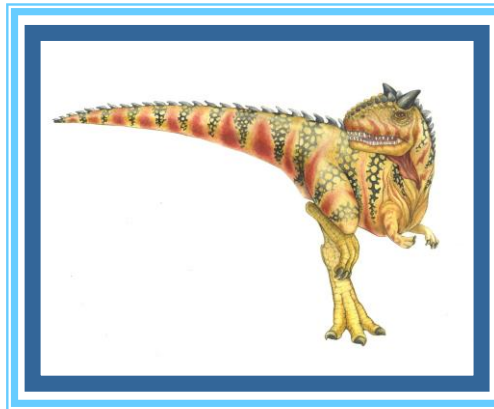
---

- ❑ Named Pipes are more powerful than ordinary pipes
- ❑ Communication is bidirectional
- ❑ No parent-child relationship is necessary between the communicating processes
- ❑ Several processes can use the named pipe for communication
- ❑ Provided on both UNIX and Windows systems



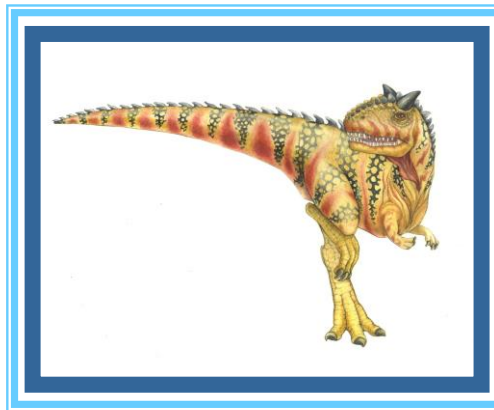
# End of Chapter 3

---



# Chapter 4: Threads

---







# Chapter 4: Threads

---

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





# Objectives

---

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





# Motivation

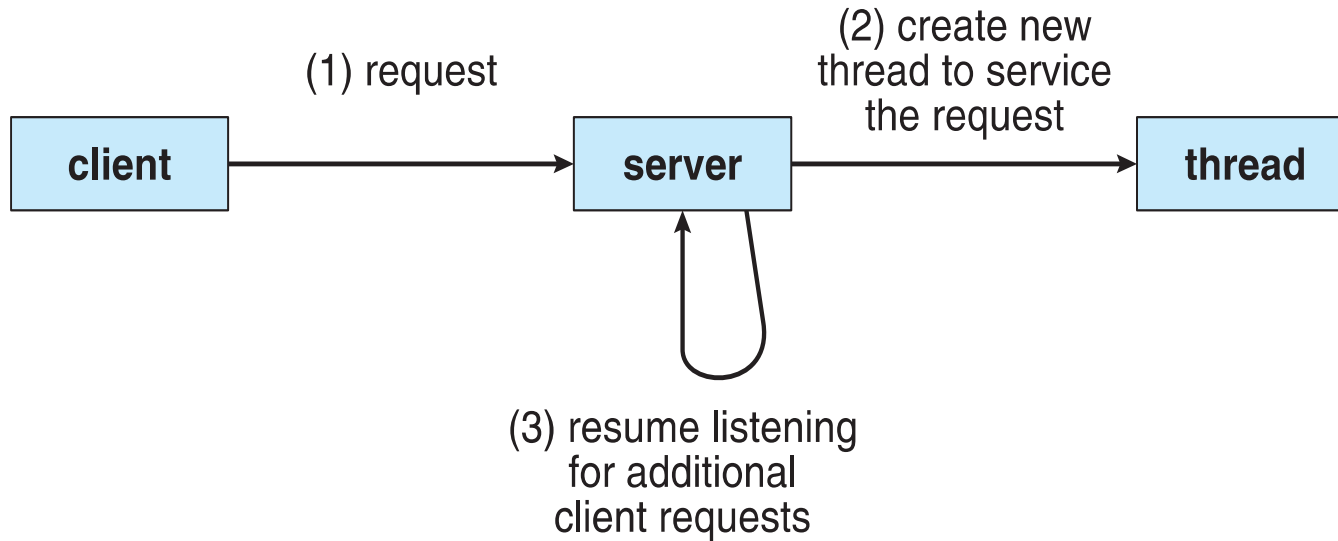
---

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





# 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

---

- **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





# Multicore Programming (Cont.)

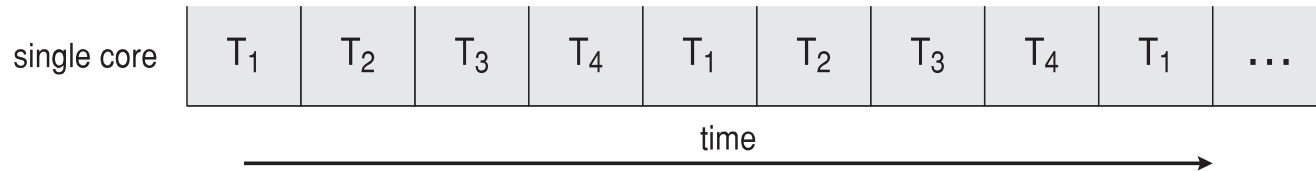
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading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



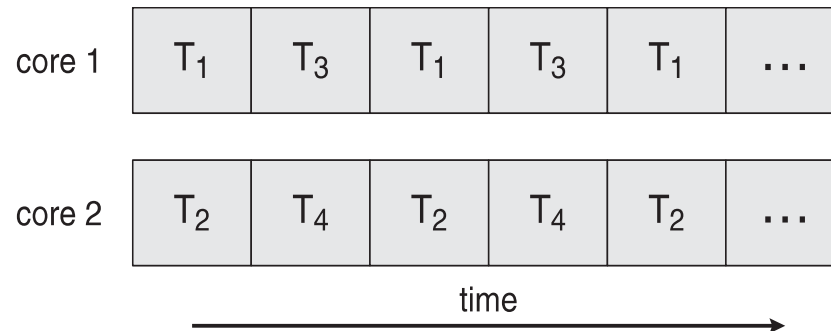


# Concurrency vs. Parallelism

## □ Concurrent execution on single-core system:



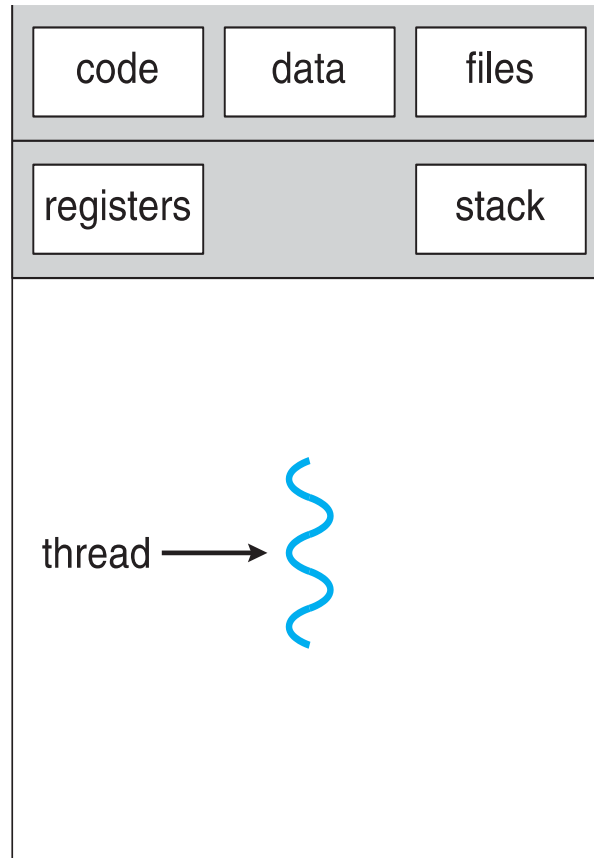
## □ Parallelism on a multi-core system:



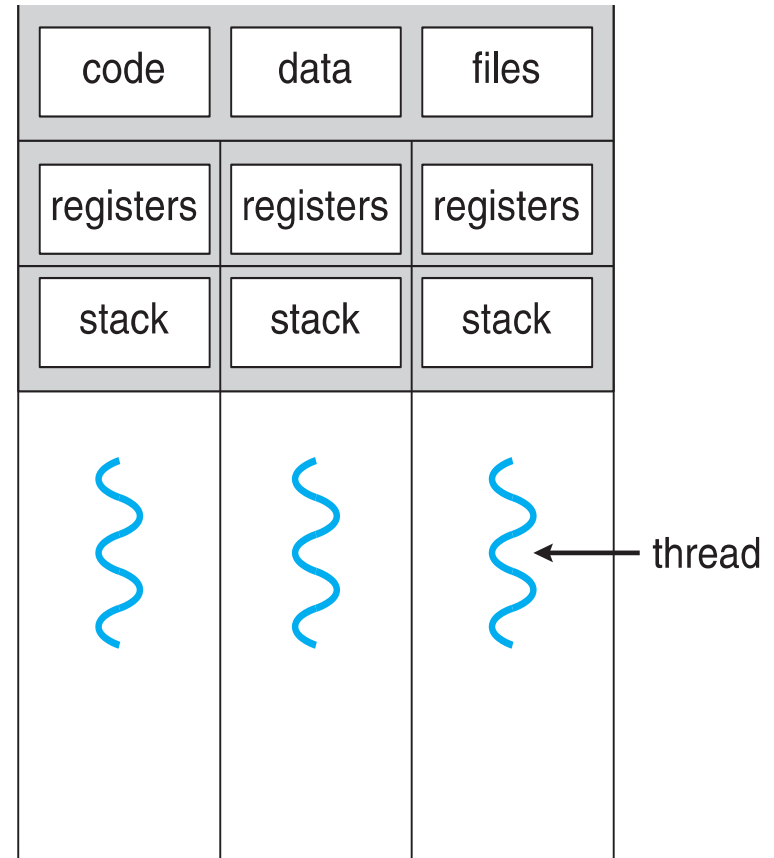




# Single and Multithreaded Processes



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
- $N$  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
- 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
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X





# Multithreading Models

---

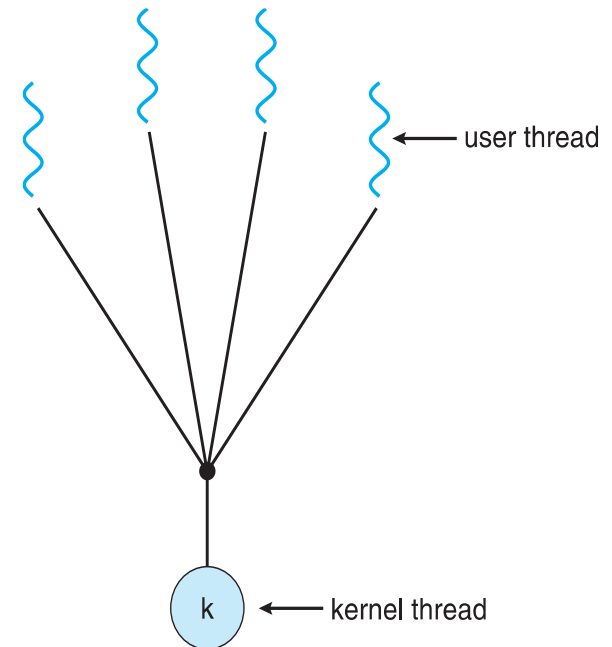
- Many-to-One
- One-to-One
- Many-to-Many





# Many-to-One

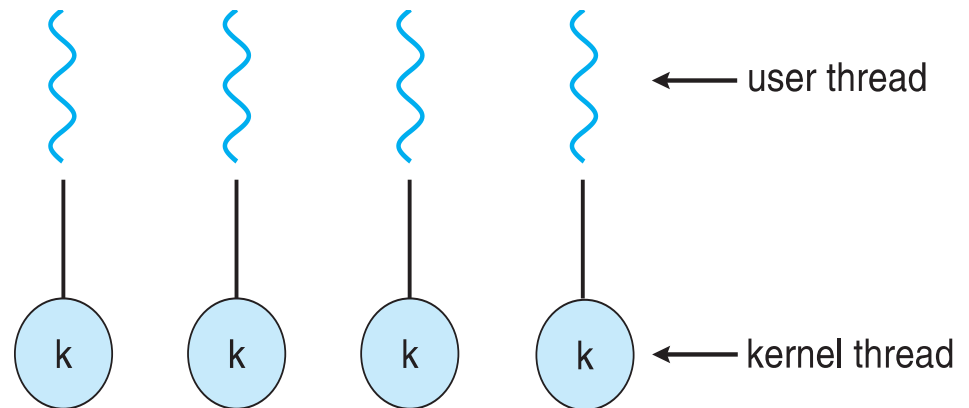
- ❑ Many user-level threads mapped to single kernel thread
- ❑ One thread blocking causes all to block
- ❑ Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- ❑ Few systems currently use this model
- ❑ Examples:
  - ❑ **Solaris Green Threads**
 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





# One-to-One

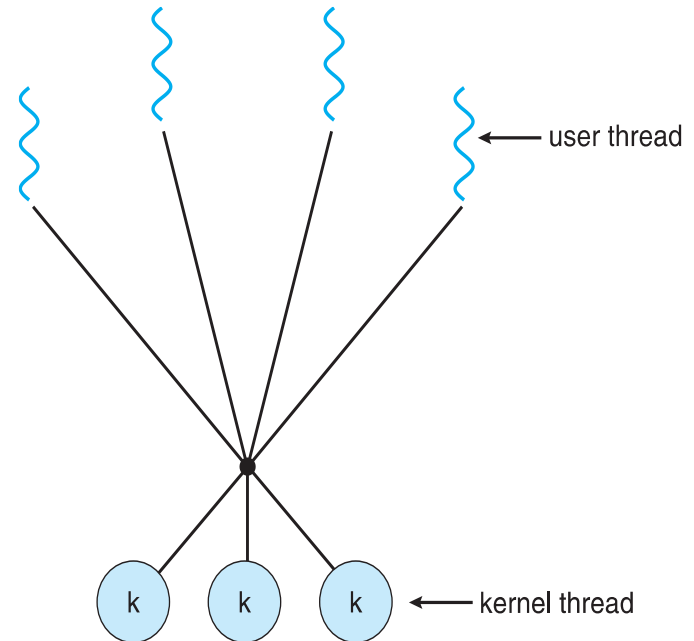
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later





# Many-to-Many Model

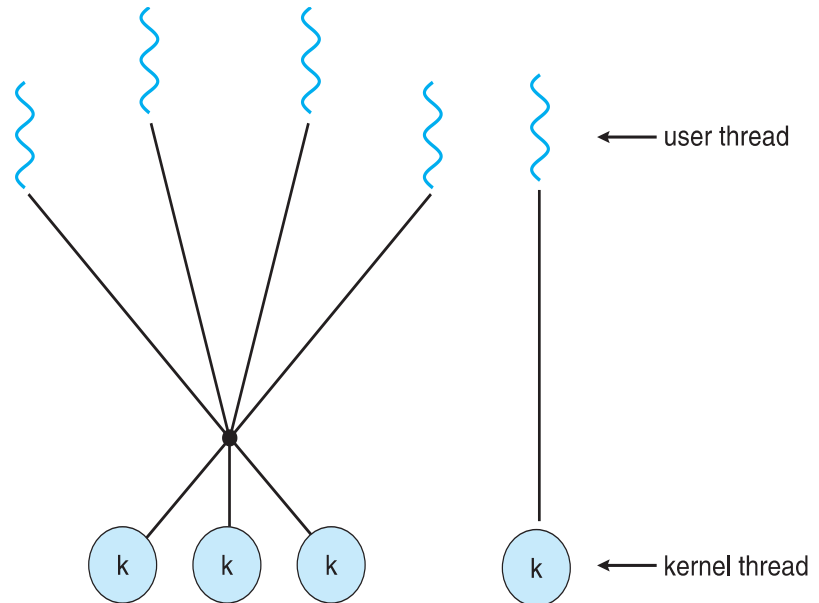
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package





# Two-level Model

- Similar to 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
  - Kernel-level library supported by the OS





# 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

---

```
#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);
```





# Windows Multithreaded C Program

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

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

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

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





# Windows Multithreaded C Program (Cont.)

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

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

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

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





# Java Threads

---

- ❑ Java threads are managed by the JVM
- ❑ Typically implemented using the threads model provided by underlying OS
- ❑ Java threads may be created by:

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

- ❑ Extending Thread class
- ❑ Implementing the Runnable interface







# 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>");
    }
}
```





# Implicit Threading

---

- ❑ Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- ❑ Creation and management of threads done by compilers and run-time libraries rather than programmers
- ❑ Three methods explored
  - ❑ Thread Pools
  - ❑ OpenMP
  - ❑ Grand Central Dispatch
- ❑ Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package





# Thread Pools

- ❑ Create a number of threads in a pool where they await work
- ❑ Advantages:
  - ❑ Usually slightly faster to service a request with an existing thread than create a new thread
  - ❑ Allows the number of threads in the application(s) to be bound to the size of the pool
  - ❑ Separating task to be performed from mechanics of creating task allows different strategies for running task
    - ▶ 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.  
     */  
}
```





# OpenMP

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

**#pragma omp parallel**

Create as many threads as there are cores

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

Run for loop in parallel

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

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

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

    /* sequential code */

    return 0;
}
```





# Grand Central Dispatch

---

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





# Grand Central Dispatch

---

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

```
dispatch_queue_t queue = dispatch_get_global_queue  
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);  
  
dispatch_async(queue, ^{ printf("I am a block."); });
```





# Threading Issues

---

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







# Semantics of `fork()` and `exec()`

---

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





# Signal Handling

- n **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- n 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
- n 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.)

---

- n 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);
```





# 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-Local Storage

---

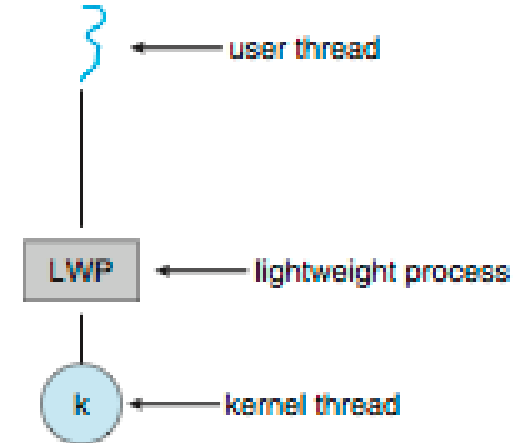
- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible 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 implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- ❑ Implements the one-to-one mapping, kernel-level
- ❑ Each thread contains
  - ❑ A thread id
  - ❑ Register set representing state of processor
  - ❑ Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - ❑ Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- ❑ The register set, stacks, and private storage area are known as the **context** of the thread





# Windows Threads (Cont.)

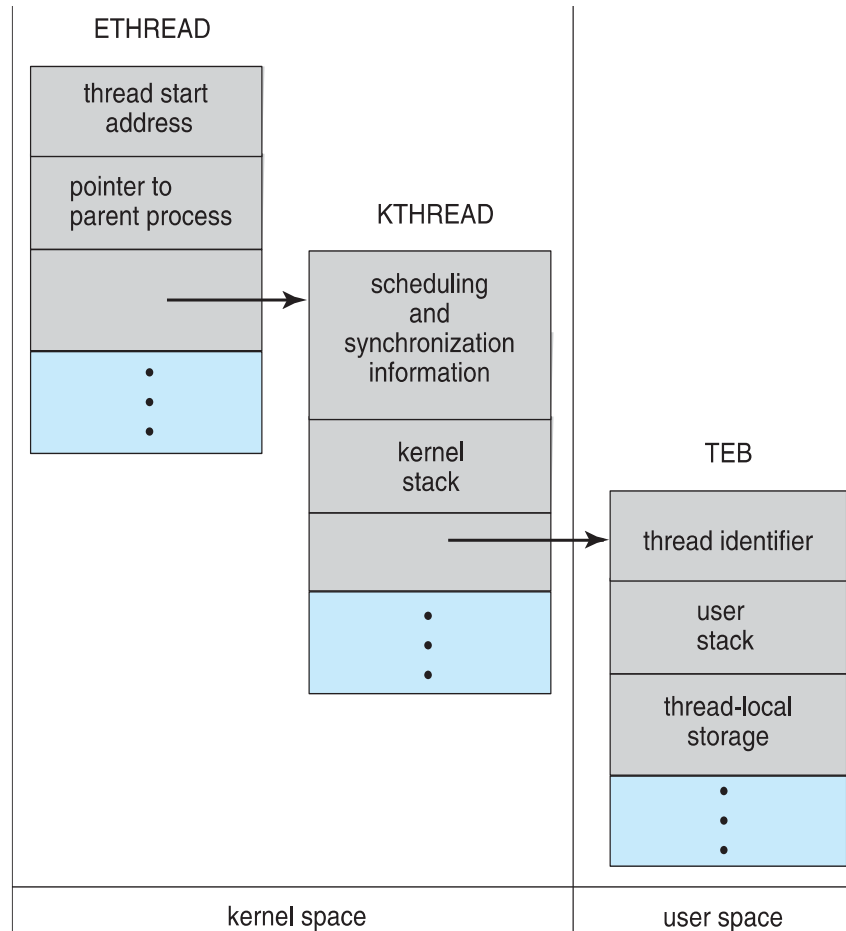
---

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





# Windows Threads Data Structures





# Linux Threads

- ❑ Linux 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

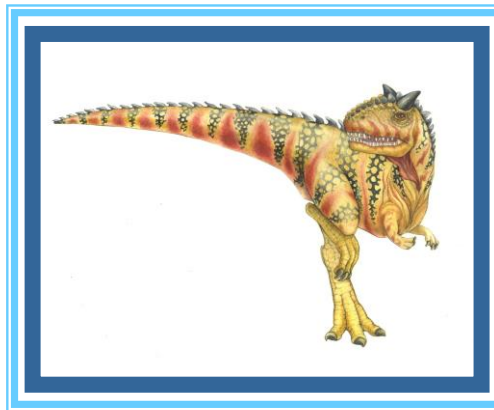
| flag          | meaning                            |
|---------------|------------------------------------|
| CLONE_FS      | File-system information is shared. |
| CLONE_VM      | The same memory space is shared.   |
| CLONE_SIGHAND | Signal handlers are shared.        |
| CLONE_FILES   | The set of open files is shared.   |

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



# End of Chapter 4

---



# Chapter 6: CPU Scheduling

---





# Chapter 6: CPU Scheduling

---

- ❑ Basic Concepts
- ❑ Scheduling Criteria
- ❑ Scheduling Algorithms
- ❑ Thread Scheduling
- ❑ Multiple-Processor Scheduling
- ❑ Real-Time CPU Scheduling
- ❑ Operating Systems Examples
- ❑ Algorithm Evaluation





# 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

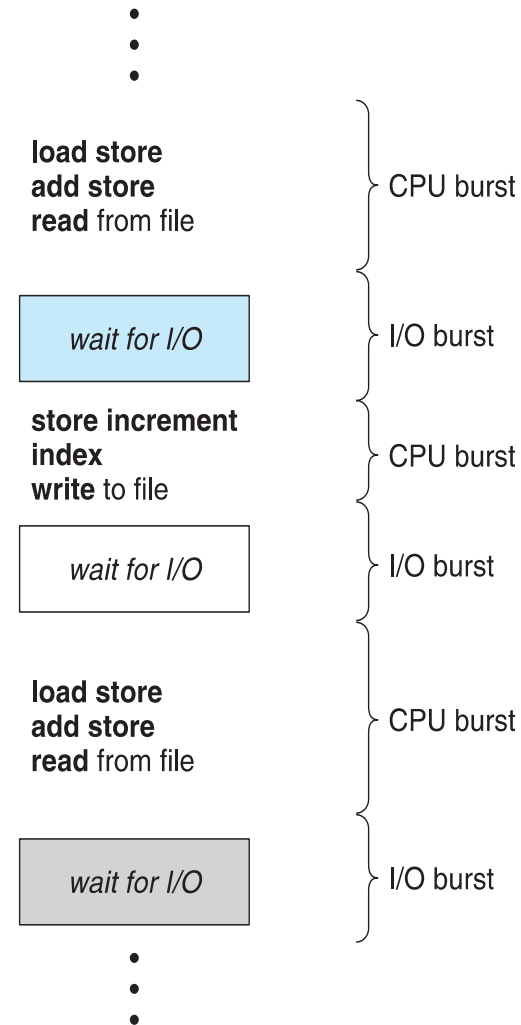






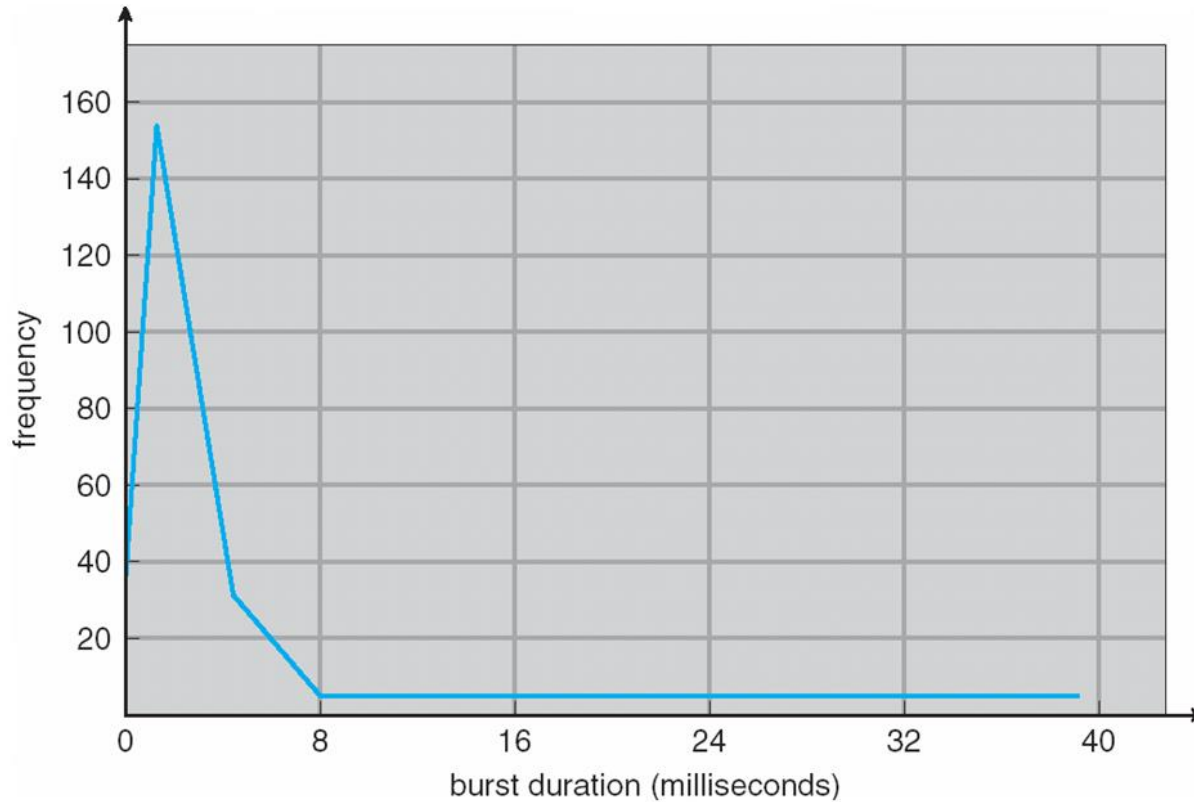
# Basic Concepts

- ❑ Maximum CPU utilization obtained with multiprogramming
- ❑ CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- ❑ **CPU burst** followed by **I/O burst**
- ❑ CPU burst distribution is of main concern





# Histogram of CPU-burst Times





# CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities





# Dispatcher

---

- ❑ Dispatcher module gives control of the CPU to the process selected by the short-term 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** – # of processes that complete their execution per time unit
- ❑ **Turnaround time** – amount of time to execute a particular process
- ❑ **Waiting time** – 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)





# Scheduling Algorithm Optimization Criteria

---

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





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

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| $P_1$          | 24                |
| $P_2$          | 3                 |
| $P_3$          | 3                 |

- Suppose that the processes arrive in the order:  $P_1$ ,  $P_2$ ,  $P_3$   
The Gantt Chart for the schedule is:



- 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) Scheduling

---

- 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
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

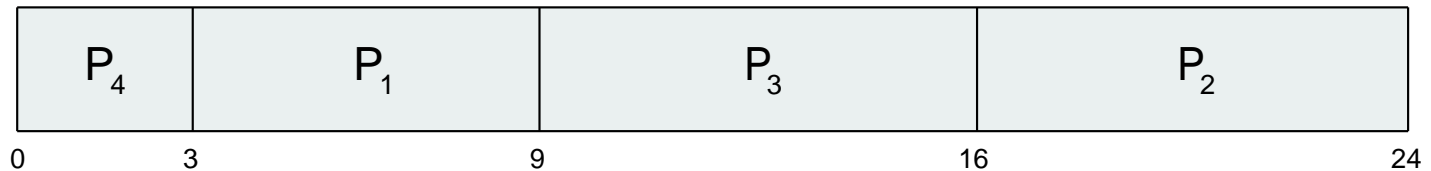




# Example of SJF

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| $P_1$          | 6                 |
| $P_2$          | 8                 |
| $P_3$          | 7                 |
| $P_4$          | 3                 |

□ SJF scheduling chart



□ Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$





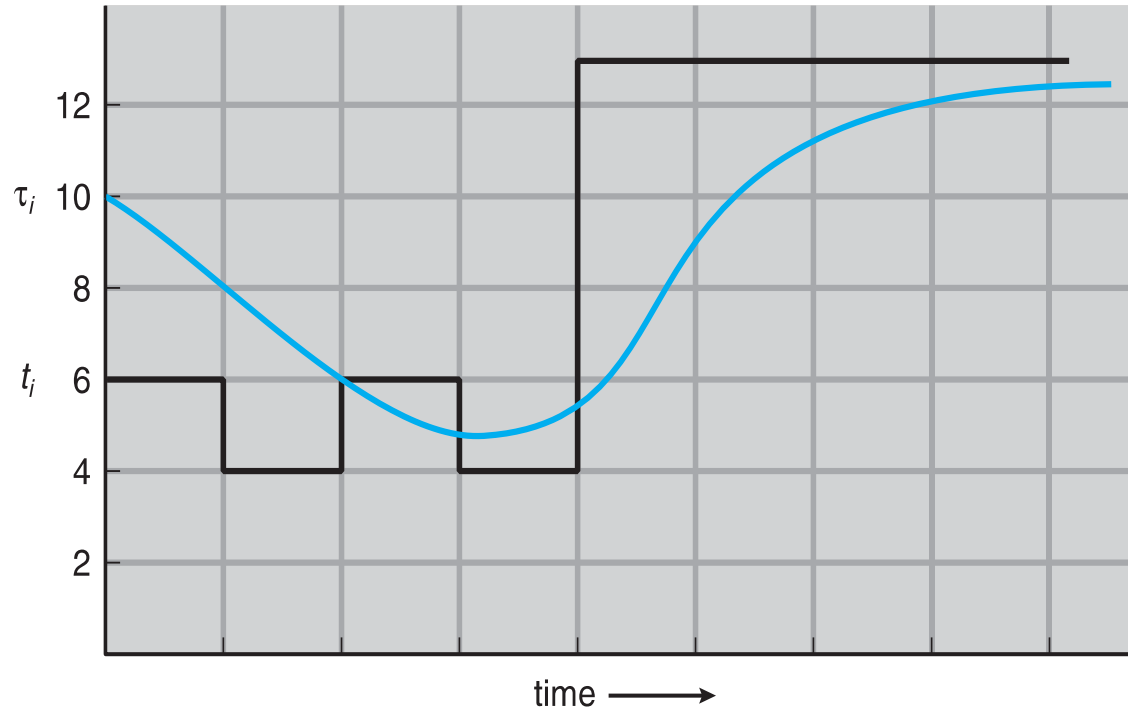
# Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  1.  $t_n$  = actual length of  $n^{th}$  CPU burst
  2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  3.  $\alpha, 0 \leq \alpha \leq 1$
  4. Define :  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ .
- Commonly,  $\alpha$  set to  $\frac{1}{2}$
- Preemptive version called **shortest-remaining-time-first**





# Prediction of the Length of the Next CPU Burst



|                      |    |   |   |   |    |    |    |     |     |
|----------------------|----|---|---|---|----|----|----|-----|-----|
| CPU burst ( $t_i$ )  | 6  | 4 | 6 | 4 | 13 | 13 | 13 | ... |     |
| "guess" ( $\tau_i$ ) | 10 | 8 | 6 | 6 | 5  | 9  | 11 | 12  | ... |





# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$ 
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} = & \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ & + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ & + (1 - \alpha)^{n+1} \tau_0\end{aligned}$$

- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor



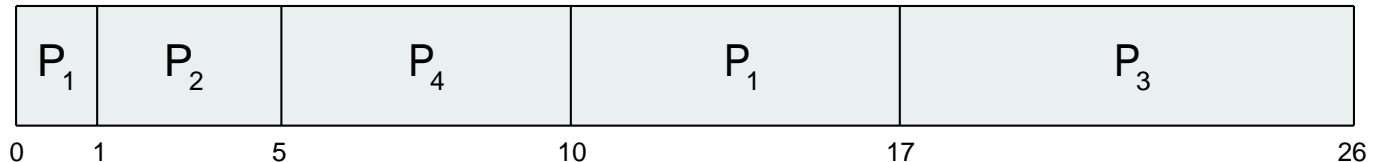


# Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

| <u>Process</u> | <u>Arrival Time</u> | <u>Burst Time</u> |
|----------------|---------------------|-------------------|
| $P_1$          | 0                   | 8                 |
| $P_2$          | 1                   | 4                 |
| $P_3$          | 2                   | 9                 |
| $P_4$          | 3                   | 5                 |

- Preemptive* SJF Gantt Chart



- Average waiting time =  $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$  msec





# Priority Scheduling

---

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
- Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process

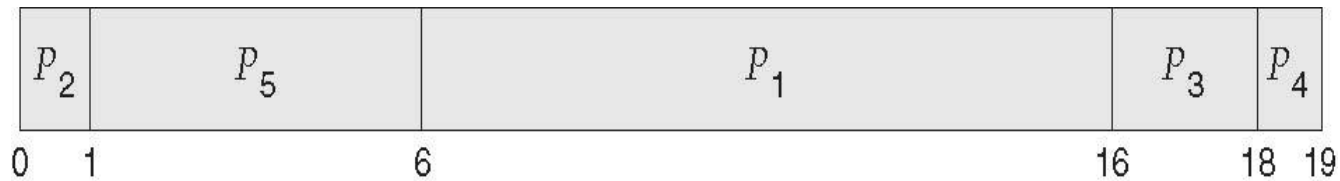




# Example of Priority Scheduling

| <u>Process</u> | <u>Burst Time</u> | <u>Priority</u> |
|----------------|-------------------|-----------------|
| $P_1$          | 10                | 3               |
| $P_2$          | 1                 | 1               |
| $P_3$          | 2                 | 4               |
| $P_4$          | 1                 | 5               |
| $P_5$          | 5                 | 2               |

## □ Priority scheduling Gantt Chart



## □ Average waiting time = 8.2 msec







# Round Robin (RR)

---

- Each process gets a small unit of CPU time (**time quantum**  $q$ ), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once. No process waits more than  $(n-1)q$  time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - $q$  large  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow q$  must be large with respect to context switch, otherwise overhead is too high

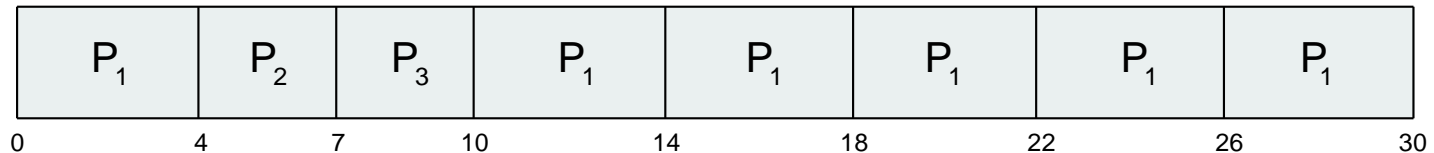




# Example of RR with Time Quantum = 4

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| $P_1$          | 24                |
| $P_2$          | 3                 |
| $P_3$          | 3                 |

□ The Gantt chart is:

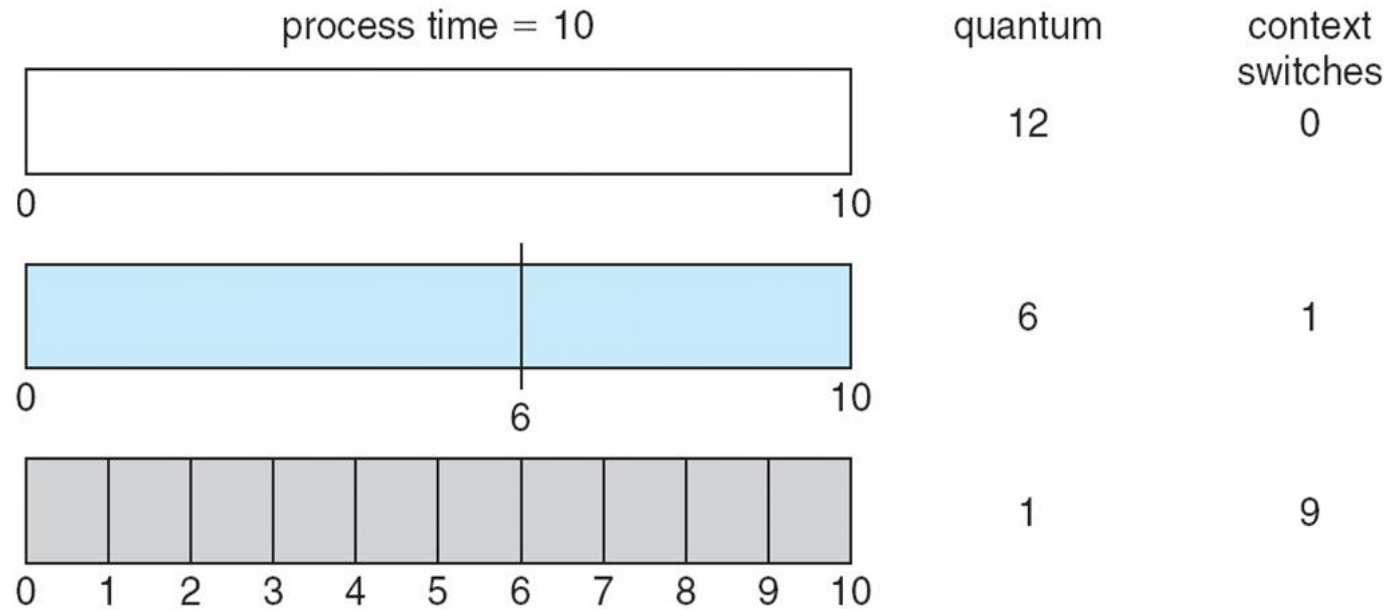


- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
- $q$  usually 10ms to 100ms, context switch < 10 usec



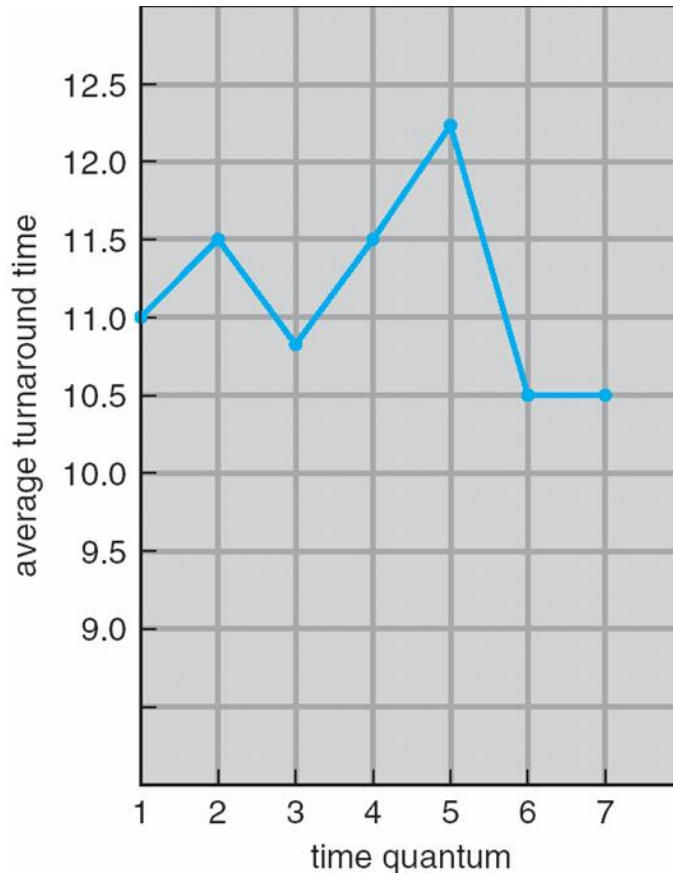


# Time Quantum and Context Switch Time





# Turnaround Time Varies With The Time Quantum



| process | time |
|---------|------|
| $P_1$   | 6    |
| $P_2$   | 3    |
| $P_3$   | 1    |
| $P_4$   | 7    |

80% of CPU bursts  
should be shorter than  $q$





# Multilevel Queue

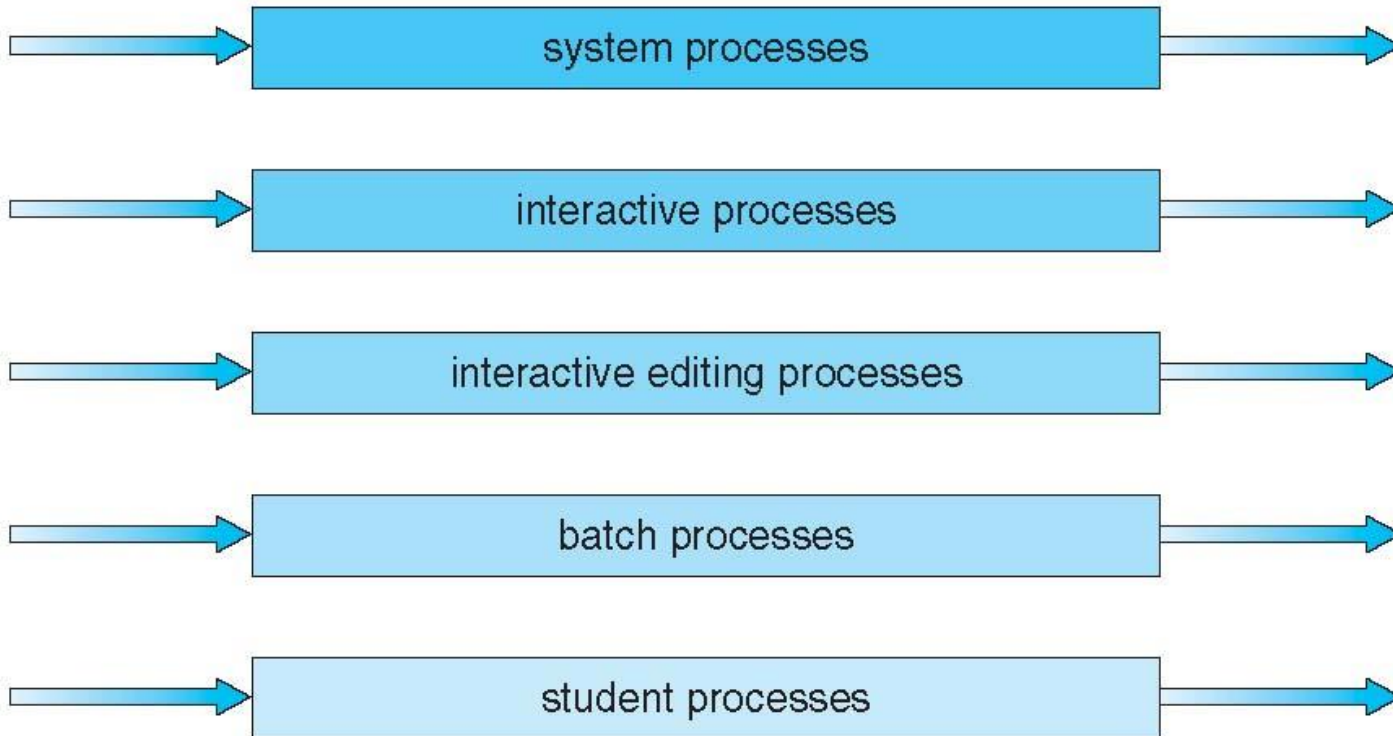
- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS





# Multilevel Queue Scheduling

highest priority



lowest priority





# Multilevel Feedback Queue

---

- ❑ A process can move between the various queues; aging can be implemented this way
- ❑ Multilevel-feedback-queue scheduler defined by the following parameters:
  - ❑ number of queues
  - ❑ scheduling algorithms for each queue
  - ❑ method used to determine when to upgrade a process
  - ❑ method used to determine when to demote a process
  - ❑ method used to determine which queue a process will enter when that process needs service



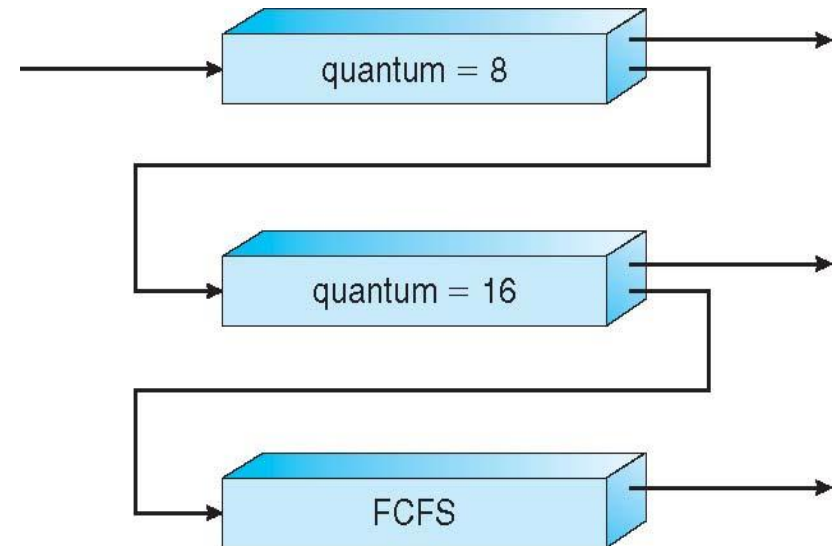


# Example of Multilevel Feedback Queue

- Three queues:
  - $Q_0$  – RR with time quantum 8 milliseconds
  - $Q_1$  – RR time quantum 16 milliseconds
  - $Q_2$  – FCFS

- Scheduling

- A new job enters queue  $Q_0$  which is served FCFS
  - ▶ When it gains CPU, job receives 8 milliseconds
  - ▶ If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$
- At  $Q_1$  job is again served FCFS and receives 16 additional milliseconds
  - ▶ If it still does not complete, it is preempted and moved to queue  $Q_2$







# Thread Scheduling

---

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system





# Pthread Scheduling

---

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD\_SCOPE\_PROCESS schedules threads using PCS scheduling
  - PTHREAD\_SCOPE\_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD\_SCOPE\_SYSTEM





# Pthread Scheduling API

---

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```





# Pthread Scheduling API

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```





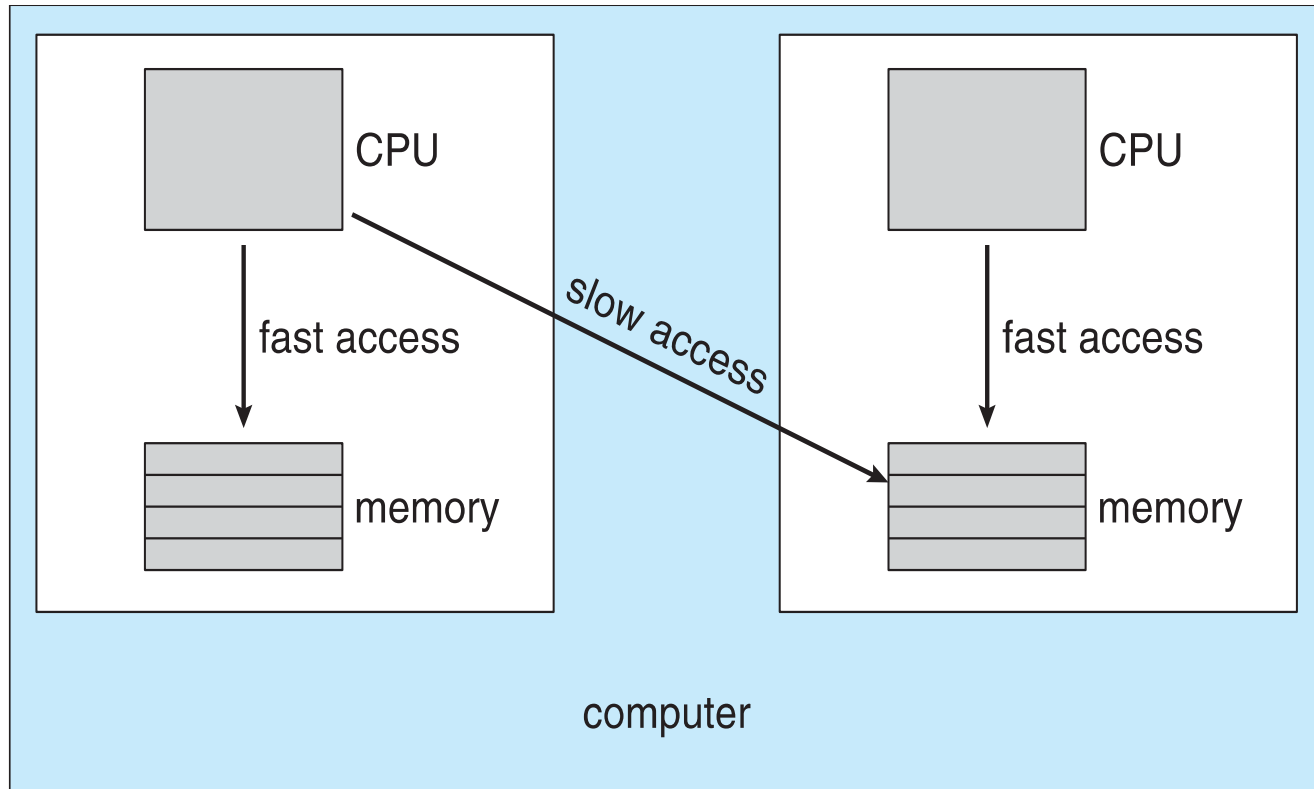
# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**





# NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity





# Multiple-Processor Scheduling – Load Balancing

---

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor





# Multicore Processors

---

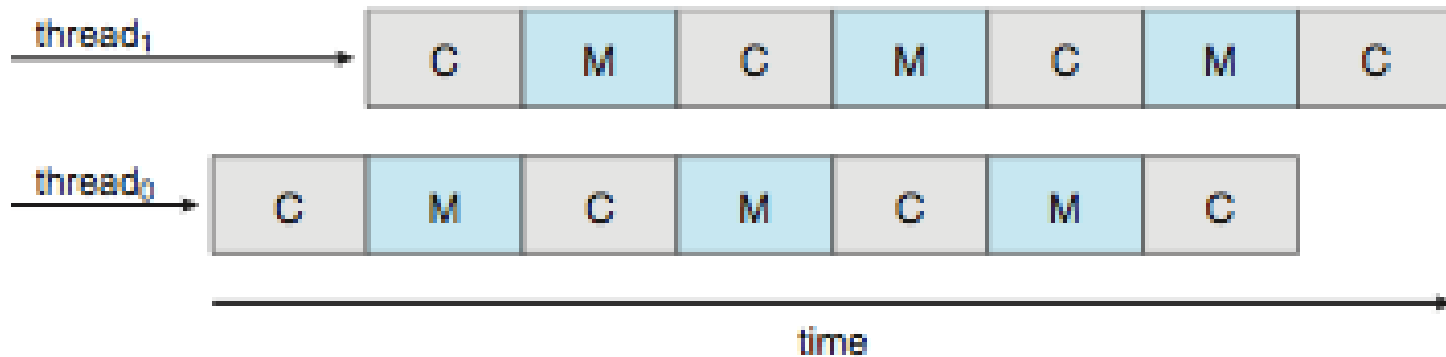
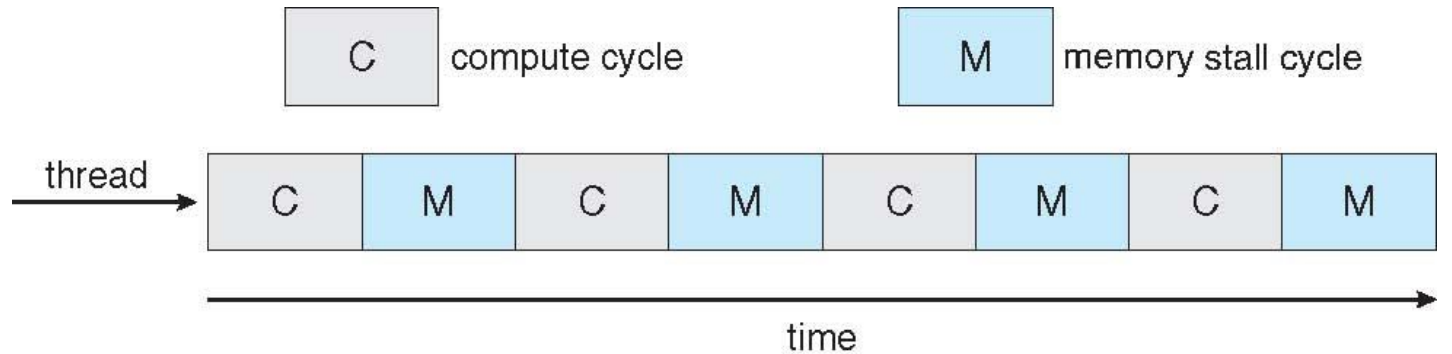
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens







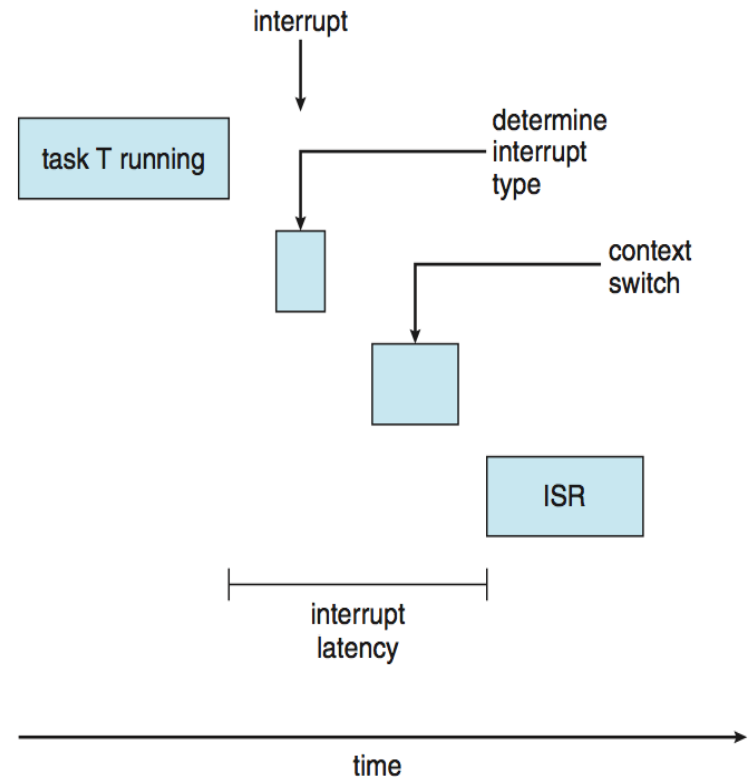
# Multithreaded Multicore System





# Real-Time CPU Scheduling

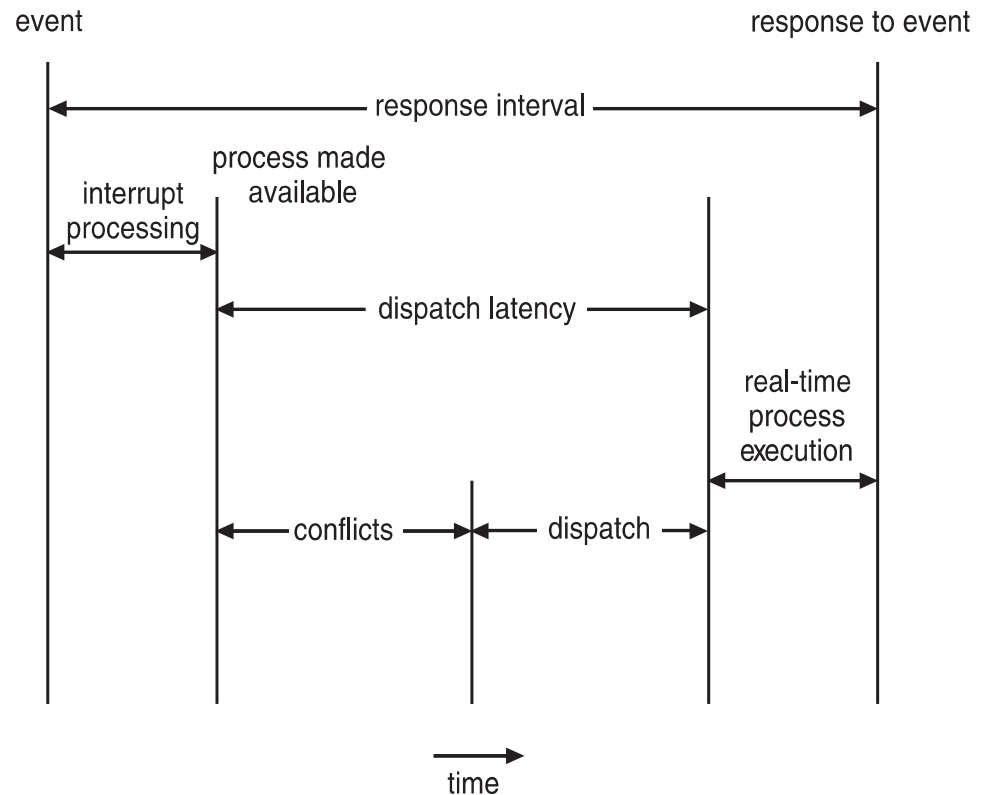
- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another





# Real-Time CPU Scheduling (Cont.)

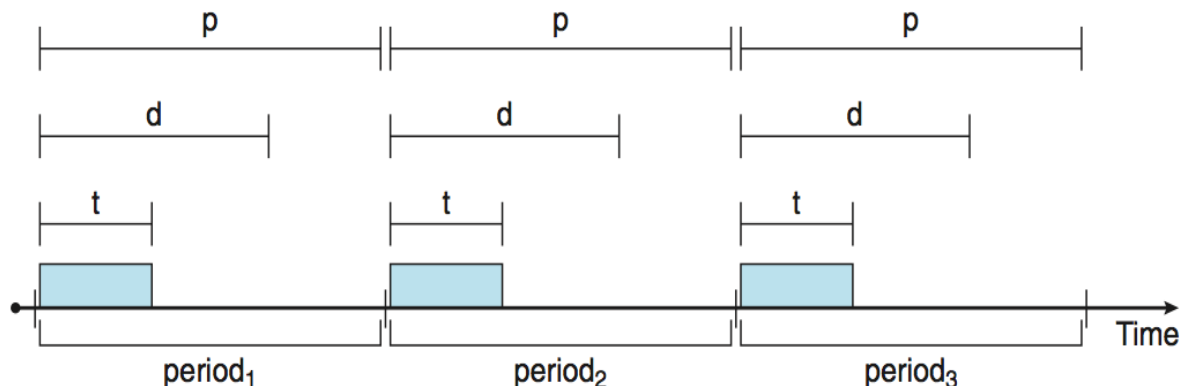
- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes





# Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time  $t$ , deadline  $d$ , period  $p$
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is  $1/p$





# Virtualization and Scheduling

---

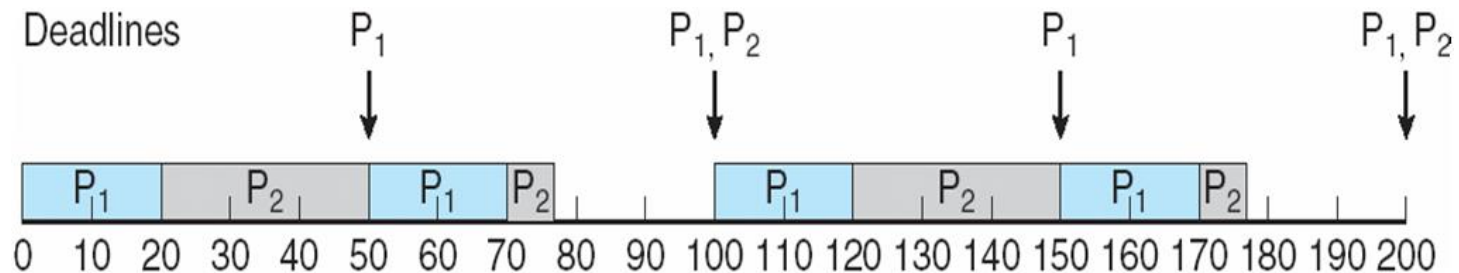
- ❑ Virtualization software schedules multiple guests onto CPU(s)
- ❑ Each guest doing its own scheduling
  - ❑ Not knowing it doesn't own the CPUs
  - ❑ Can result in poor response time
  - ❑ Can effect time-of-day clocks in guests
- ❑ Can undo good scheduling algorithm efforts of guests





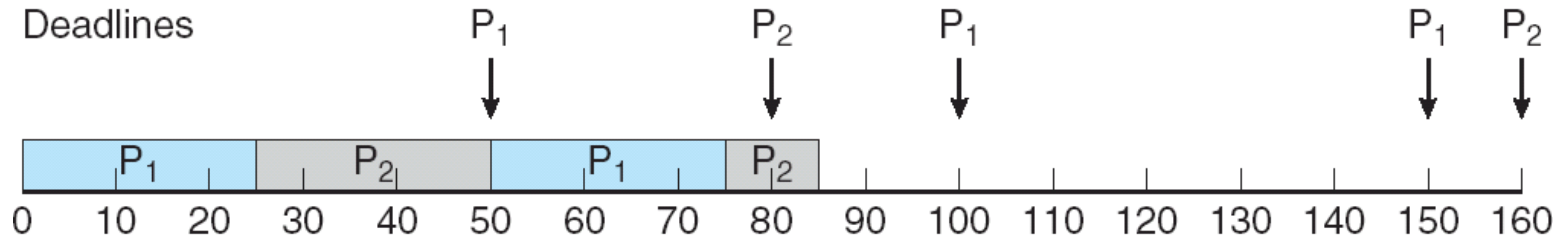
# Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- $P_1$  is assigned a higher priority than  $P_2$ .





# Missed Deadlines with Rate Monotonic Scheduling



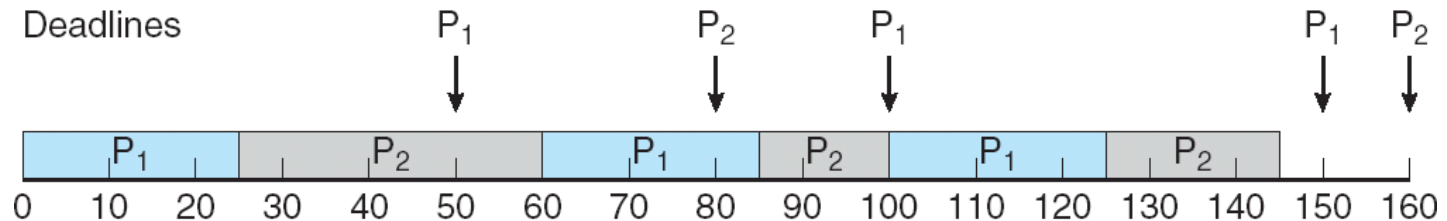


# Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

the earlier the deadline, the higher the priority;

the later the deadline, the lower the priority







# Proportional Share Scheduling

---

- $T$  shares are allocated among all processes in the system
- An application receives  $N$  shares where  $N < T$
- This ensures each application will receive  $N / T$  of the total processor time





# POSIX Real-Time Scheduling

- n The POSIX.1b standard
- n API provides functions for managing real-time threads
- n Defines two scheduling classes for real-time threads:
  1. SCHED\_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. SCHED\_RR - similar to SCHED\_FIFO except time-slicing occurs for threads of equal priority
- n Defines two functions for getting and setting scheduling policy:
  1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
  2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`





# POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t_tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Unable to get policy.\n");
    else {
        if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR) printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
    }
}
```





# POSIX Real-Time Scheduling API (Cont.)

```
/* set the scheduling policy - FIFO, RR, or OTHER */
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.\n");
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}
```





# Operating System Examples

---

- ❑ Linux scheduling
- ❑ Windows scheduling
- ❑ Solaris scheduling





# Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order  $O(1)$  scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger  $q$
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - ▶ Two priority arrays (active, expired)
    - ▶ Tasks indexed by priority
    - ▶ When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes





# Linux Scheduling in Version 2.6.23 +

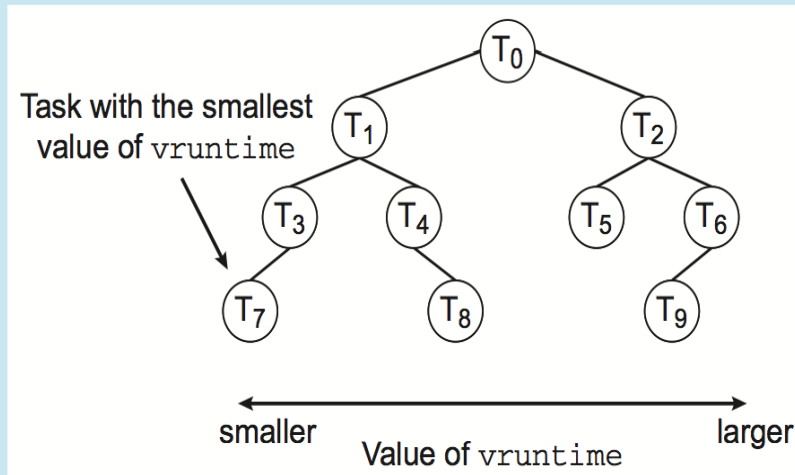
- ❑ **Completely Fair Scheduler (CFS)**
- ❑ **Scheduling classes**
  - ❑ Each has specific priority
  - ❑ Scheduler picks highest priority task in highest scheduling class
  - ❑ Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - ❑ 2 scheduling classes included, others can be added
    1. default
    2. real-time
- ❑ Quantum calculated based on **nice value** from -20 to +19
  - ❑ Lower value is higher priority
  - ❑ Calculates **target latency** – interval of time during which task should run at least once
  - ❑ Target latency can increase if say number of active tasks increases
- ❑ CFS scheduler maintains per task **virtual run time** in variable **vruntime**
  - ❑ Associated with decay factor based on priority of task – lower priority is higher decay rate
  - ❑ Normal default priority yields virtual run time = actual run time
- ❑ To decide next task to run, scheduler picks task with lowest virtual run time





# CFS Performance

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require  $O(\lg N)$  operations (where  $N$  is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

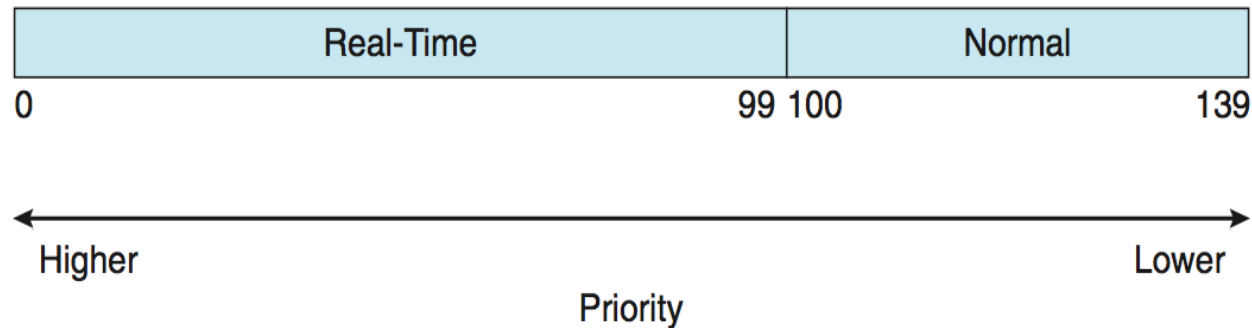






# Linux Scheduling (Cont.)

- ❑ Real-time scheduling according to POSIX.1b
  - ❑ Real-time tasks have static priorities
- ❑ Real-time plus normal map into global priority scheme
- ❑ Nice value of -20 maps to global priority 100
- ❑ Nice value of +19 maps to priority 139





# Windows Scheduling

---

- ❑ Windows uses priority-based preemptive scheduling
- ❑ Highest-priority thread runs next
- ❑ **Dispatcher** is scheduler
- ❑ Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- ❑ Real-time threads can preempt non-real-time
- ❑ 32-level priority scheme
- ❑ **Variable class** is 1-15, **real-time class** is 16-31
- ❑ Priority 0 is memory-management thread
- ❑ Queue for each priority
- ❑ If no run-able thread, runs **idle thread**





# Windows Priority Classes

---

- Win32 API identifies several priority classes to which a process can belong
  - REALTIME\_PRIORITY\_CLASS, HIGH\_PRIORITY\_CLASS, ABOVE\_NORMAL\_PRIORITY\_CLASS, NORMAL\_PRIORITY\_CLASS, BELOW\_NORMAL\_PRIORITY\_CLASS, IDLE\_PRIORITY\_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority
  - TIME\_CRITICAL, HIGHEST, ABOVE\_NORMAL, NORMAL, BELOW\_NORMAL, LOWEST, IDLE
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base





# Windows Priority Classes (Cont.)

---

- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost
- Windows 7 added **user-mode scheduling (UMS)**
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework





# Windows Priorities

|               | real-time | high | above normal | normal | below normal | idle priority |
|---------------|-----------|------|--------------|--------|--------------|---------------|
| time-critical | 31        | 15   | 15           | 15     | 15           | 15            |
| highest       | 26        | 15   | 12           | 10     | 8            | 6             |
| above normal  | 25        | 14   | 11           | 9      | 7            | 5             |
| normal        | 24        | 13   | 10           | 8      | 6            | 4             |
| below normal  | 23        | 12   | 9            | 7      | 5            | 3             |
| lowest        | 22        | 11   | 8            | 6      | 4            | 2             |
| idle          | 16        | 1    | 1            | 1      | 1            | 1             |





# Solaris

---

- ❑ Priority-based scheduling
- ❑ Six classes available
  - ❑ Time sharing (default) (TS)
  - ❑ Interactive (IA)
  - ❑ Real time (RT)
  - ❑ System (SYS)
  - ❑ Fair Share (FSS)
  - ❑ Fixed priority (FP)
- ❑ Given thread can be in one class at a time
- ❑ Each class has its own scheduling algorithm
- ❑ Time sharing is multi-level feedback queue
  - ❑ Loadable table configurable by sysadmin





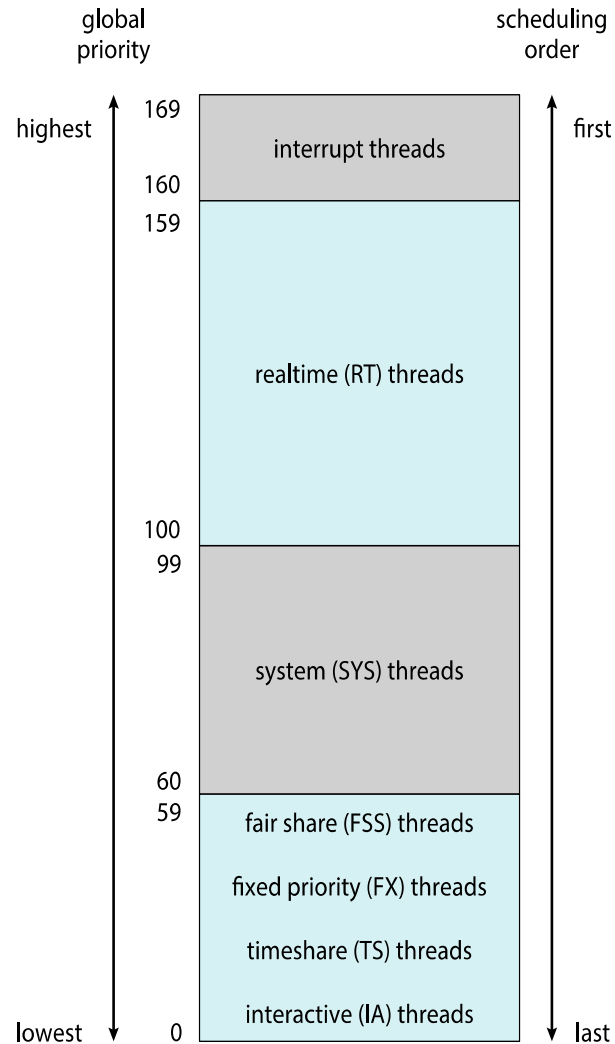
# Solaris Dispatch Table

| priority | time quantum | time quantum expired | return from sleep |
|----------|--------------|----------------------|-------------------|
| 0        | 200          | 0                    | 50                |
| 5        | 200          | 0                    | 50                |
| 10       | 160          | 0                    | 51                |
| 15       | 160          | 5                    | 51                |
| 20       | 120          | 10                   | 52                |
| 25       | 120          | 15                   | 52                |
| 30       | 80           | 20                   | 53                |
| 35       | 80           | 25                   | 54                |
| 40       | 40           | 30                   | 55                |
| 45       | 40           | 35                   | 56                |
| 50       | 40           | 40                   | 58                |
| 55       | 40           | 45                   | 58                |
| 59       | 20           | 49                   | 59                |





# Solaris Scheduling







# Solaris Scheduling (Cont.)

---

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR





# Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| $P_1$          | 10                |
| $P_2$          | 29                |
| $P_3$          | 3                 |
| $P_4$          | 7                 |
| $P_5$          | 12                |

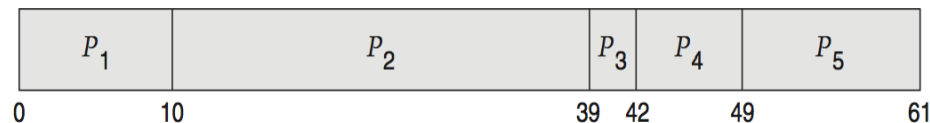




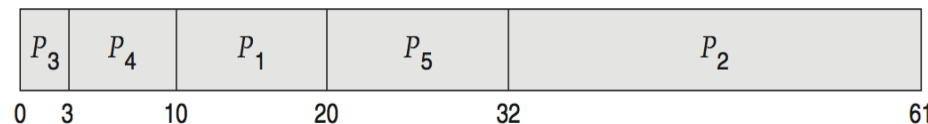
# Deterministic Evaluation

- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs

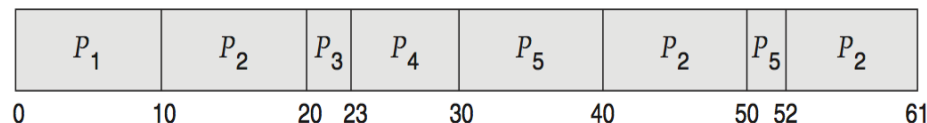
FCS is 28ms:



Non-preemptive SFJ is 13ms:



RR is 23ms:





# Queueing Models

---

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc





# Little' s Formula

---

- $n$  = average queue length
- $W$  = average waiting time in queue
- $\lambda$  = average arrival rate into queue
- Little' s law – in steady state, processes leaving queue must equal processes arriving, thus:  
$$n = \lambda \times W$$
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds





# Simulations

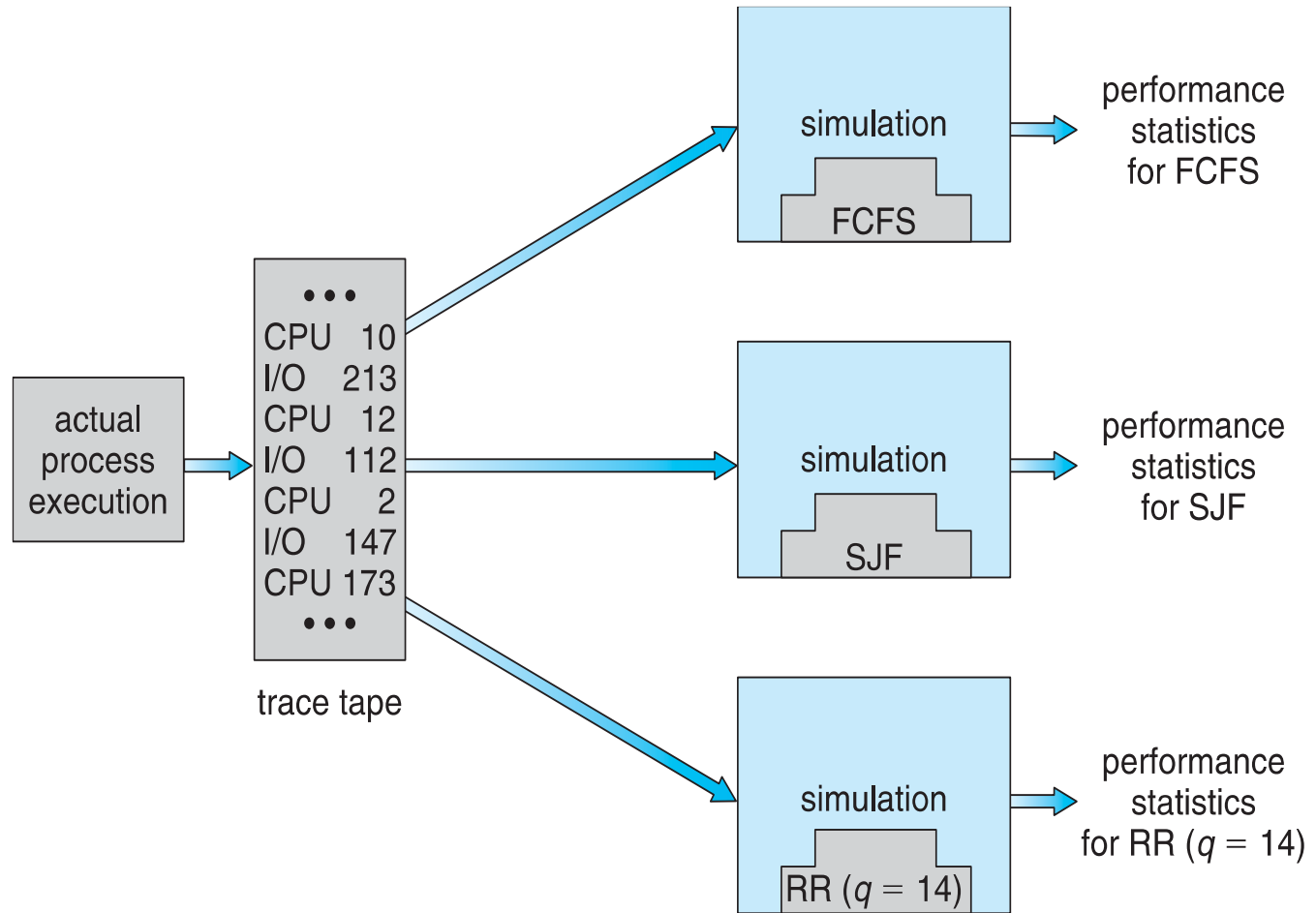
---

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - ▶ Random number generator according to probabilities
    - ▶ Distributions defined mathematically or empirically
    - ▶ Trace tapes record sequences of real events in real systems





# Evaluation of CPU Schedulers by Simulation





# Implementation

---

- ❑ Even simulations have limited accuracy
- ❑ Just implement new scheduler and test in real systems
  - ❑ High cost, high risk
  - ❑ Environments vary
- ❑ Most flexible schedulers can be modified per-site or per-system
- ❑ Or APIs to modify priorities
- ❑ But again environments vary





# End of Chapter 6

---

