



# Operating System

- Overview
- Process    进程
- Thread    线程
- Process scheduling 调度
- Concurrency 并发
- Memory management





---

A. Silberchatz, G. Gagne, P. Galvin, Operating System Concepts, 10th Edition, 2018.



# Operating System

## Chapter 1:





# Chapter 1: Introduction

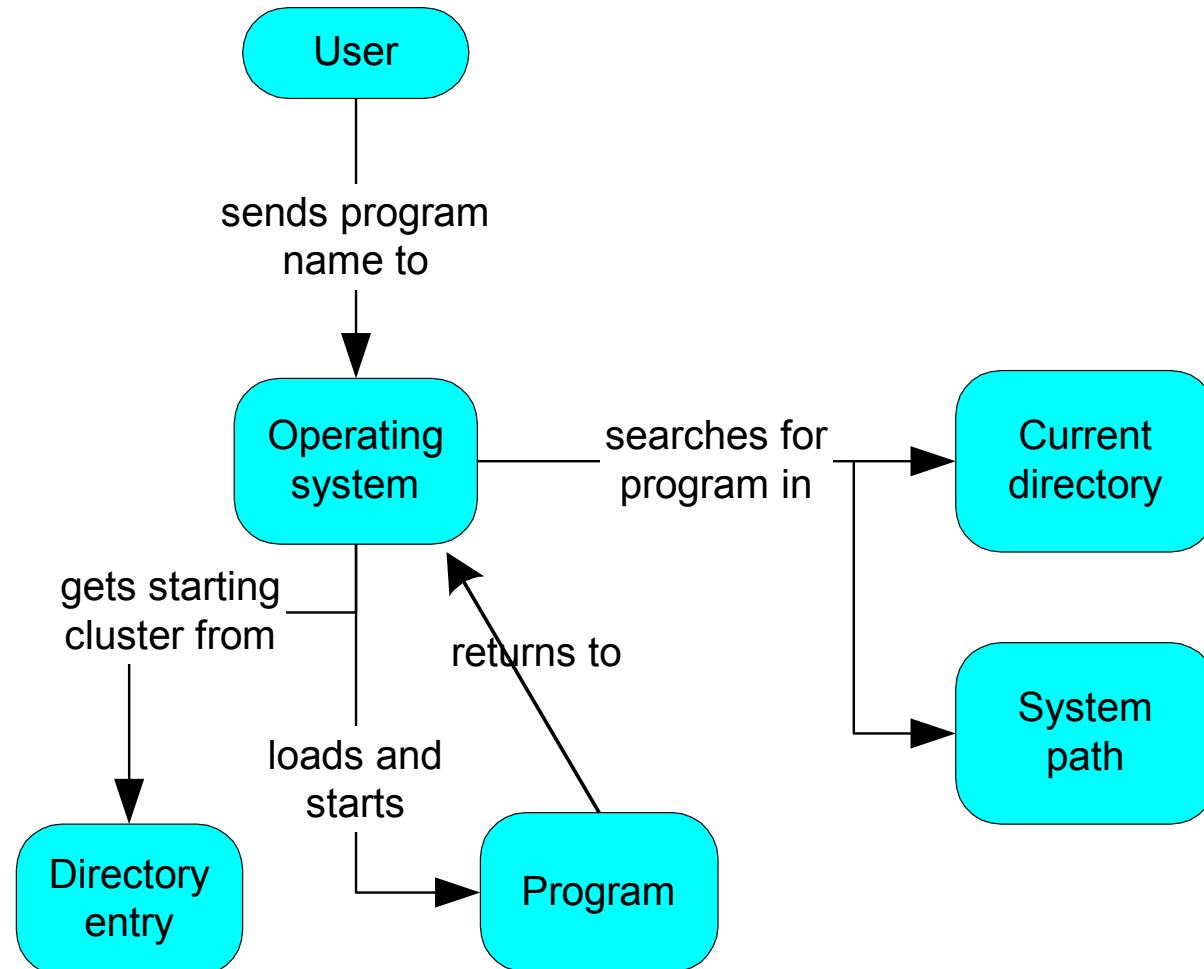
---

- What Operating Systems Do
- Computer-System Organization
- Operating-System Operations
- OS Management
  - Process Management
  - Memory Management
  - Storage Management
  - Protection and Security





# How a Program Runs





# What is an Operating System?

- A **program** that acts as an intermediary between a user of a computer and the computer hardware
- Operating means management and control on computer
- Operating system goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient to use
  - Use the computer hardware in an efficient manner





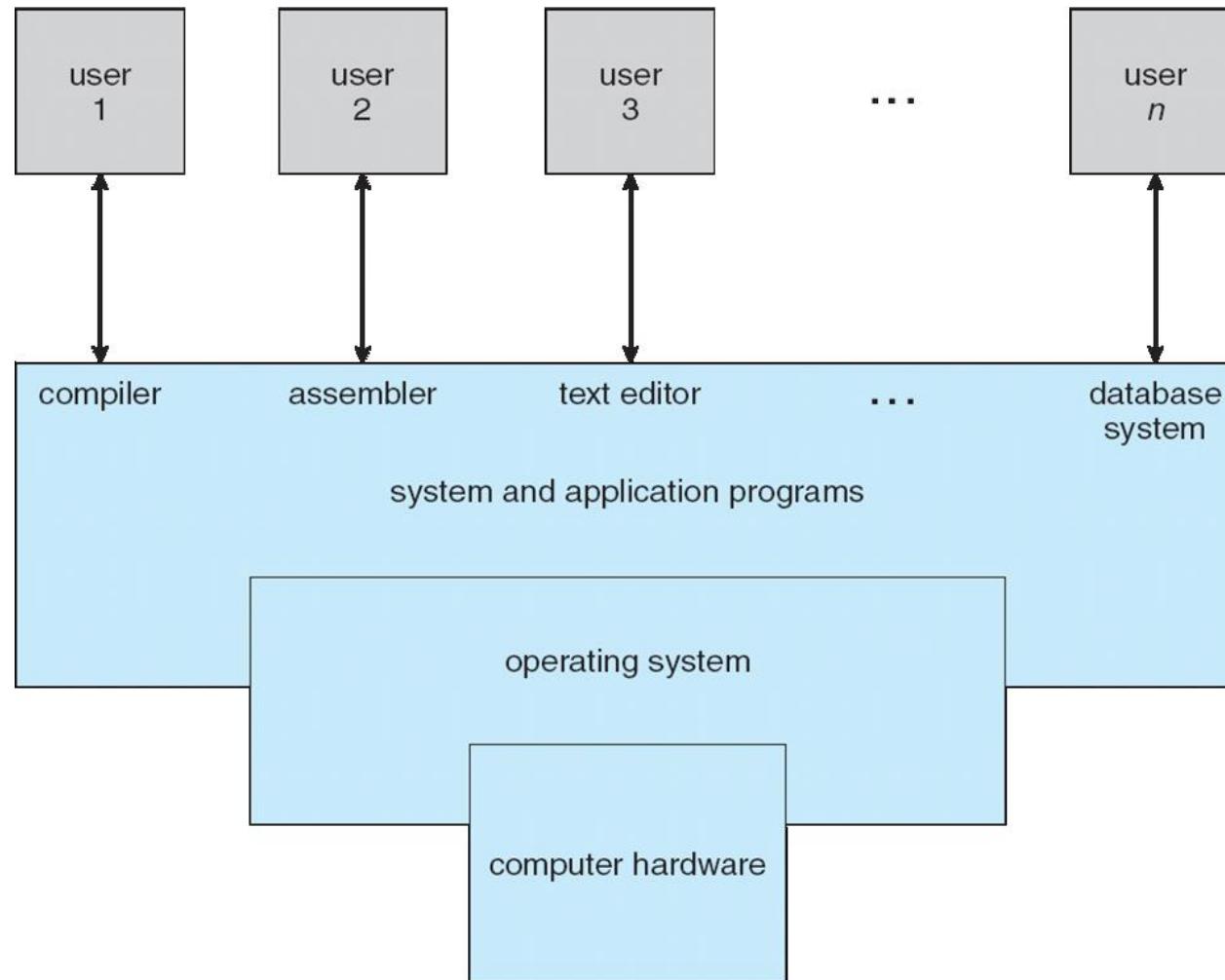
# Computer System Structure

- Computer system can be divided into four components:
  - Hardware – provides basic computing resources
    - ▶ CPU, memory, I/O devices
  - Operating system (System programs)
    - ▶ Controls and coordinates use of hardware among various applications and users
  - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
    - ▶ Word processors, compilers, web browsers, database systems, video games
  - Users
    - ▶ People, machines, other computers





# Four Components of a Computer System





# Computer Startup

- **bootstrap program** is loaded at power-up or reboot
  - Typically stored in ROM or EPROM, generally known as **firmware 固件**
  - Initializes all aspects of system
  - Loads operating system kernel and starts execution





# Operating System Operation

- OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer





# OS management

- Process 进程
- Memory 内存
- Storage 外存
- I/O 输入、输出设备
- Security and protection 安全与保护





# Process Management

---

- A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources.
- Process has one **program counter** specifying location of next instruction.
- Typically system has many processes, concurrently on CPU.
  - Concurrency by multiplexing the CPUs among the processes





# Process Management Activities

---

The operating system is responsible for the following activities in connection with process(进程) management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling





# Memory Management

---

- All data and all instructions in memory in order to execute
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed





# Storage Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - ▶ Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - ▶ Creating and deleting files and directories
    - ▶ Primitives to manipulate files and dirs
    - ▶ Mapping files onto secondary storage
    - ▶ Backup files onto stable (non-volatile) storage media





# Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time.
- Proper management is of central importance.
- OS activities:
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed
  - Varies between WORM (write-once, read-many-times) and RW (read-write)





# I/O Subsystem

- One purpose of OS is to hide peculiarities of hardware devices from the user.
- I/O subsystem responsible for:
  - Memory management of I/O including
    - ▶ buffering (storing data temporarily while it is being transferred),
    - ▶ caching (storing parts of data in faster storage for performance),
    - ▶ spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices





# Protection and Security

- **Protection** – any mechanism for controlling access of processes or users to resources defined by the OS.
- **Security** – defense of the system against internal and external attacks.
  - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (**group ID**) allows set of users to be defined and controlled, then also associated with each process, file
  - **Privilege escalation** allows user to change to effective ID with more rights





# Chapter 2: Operating-System Structures

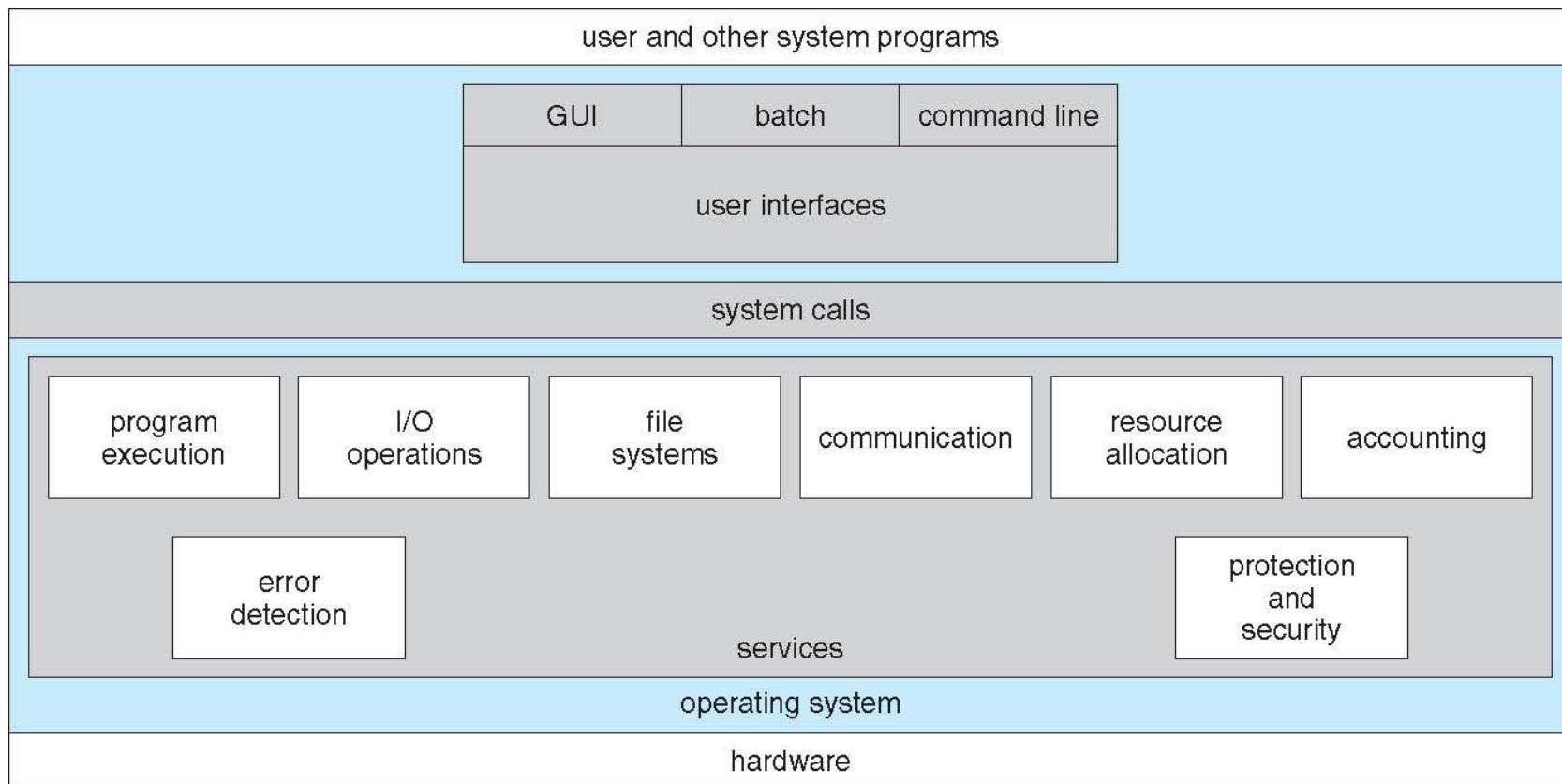
- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Operating System Generation
- System Boot





# A View of Operating System Services

## Structures





# Operating System Services

---

- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (UI)
    - ▶ Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
  - **File-system manipulation** - A programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.





# Operating System Services (Cont.)

- One set of operating-system services provides functions that are helpful to the user (cont.):
  - **Communications** – Processes may exchange information, on the same computer or between computers over a network
    - ▶ Communications may be via shared memory or through message passing (packets moved by the OS)
  - **Error detection** – OS needs to be constantly aware of possible errors
    - ▶ May occur in the CPU and memory hardware, in I/O devices, in user program
    - ▶ For each type of error, OS should take the appropriate action to ensure correct and consistent computing
    - ▶ Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system





# Operating System Services (Cont.)

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
  - **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - ▶ Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **Accounting** - To keep track of which users use how much and what kinds of computer resources
  - **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - ▶ **Protection** involves ensuring that all access to system resources is controlled
    - ▶ **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
    - ▶ If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.





# User Operating System Interface - CLI

- Command Line Interface (CLI) or **command interpreter** allows direct command entry.
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and executes it
    - ▶ Sometimes commands built-in, sometimes just names of programs





# User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
- Many systems now include both **CLI and GUI** interfaces
  - Microsoft Windows is GUI with CLI “command” shell
  - Apple Mac OS X as “Aqua” GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)





# Bourne Shell Command Interpreter

```
Terminal
File Edit View Terminal Tabs Help
fd0      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0    0
sd0      0.0    0.2    0.0    0.2    0.0    0.0    0.4    0    0    0
sd1      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0    0    0
                           extended device statistics
device   r/s    w/s    kr/s   kw/s   wait   activ  svc_t  %w   %b
fd0      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0    0    0
sd0      0.6    0.0    38.4   0.0    0.0    0.0    8.2    0    0    0
sd1      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0    0    0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# w
4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 8.66
User     tty          login@  idle   JCPU   PCPU what
root     console      15Jun07 18days   1        /usr/bin/ssh-agent -- /usr/bi
n/d
root     pts/3         15Jun07           18        4   w
root     pts/4         15Jun07           18        4   w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-/var/tmp/system-contents/scripts)#

```





# The Mac OS X GUI





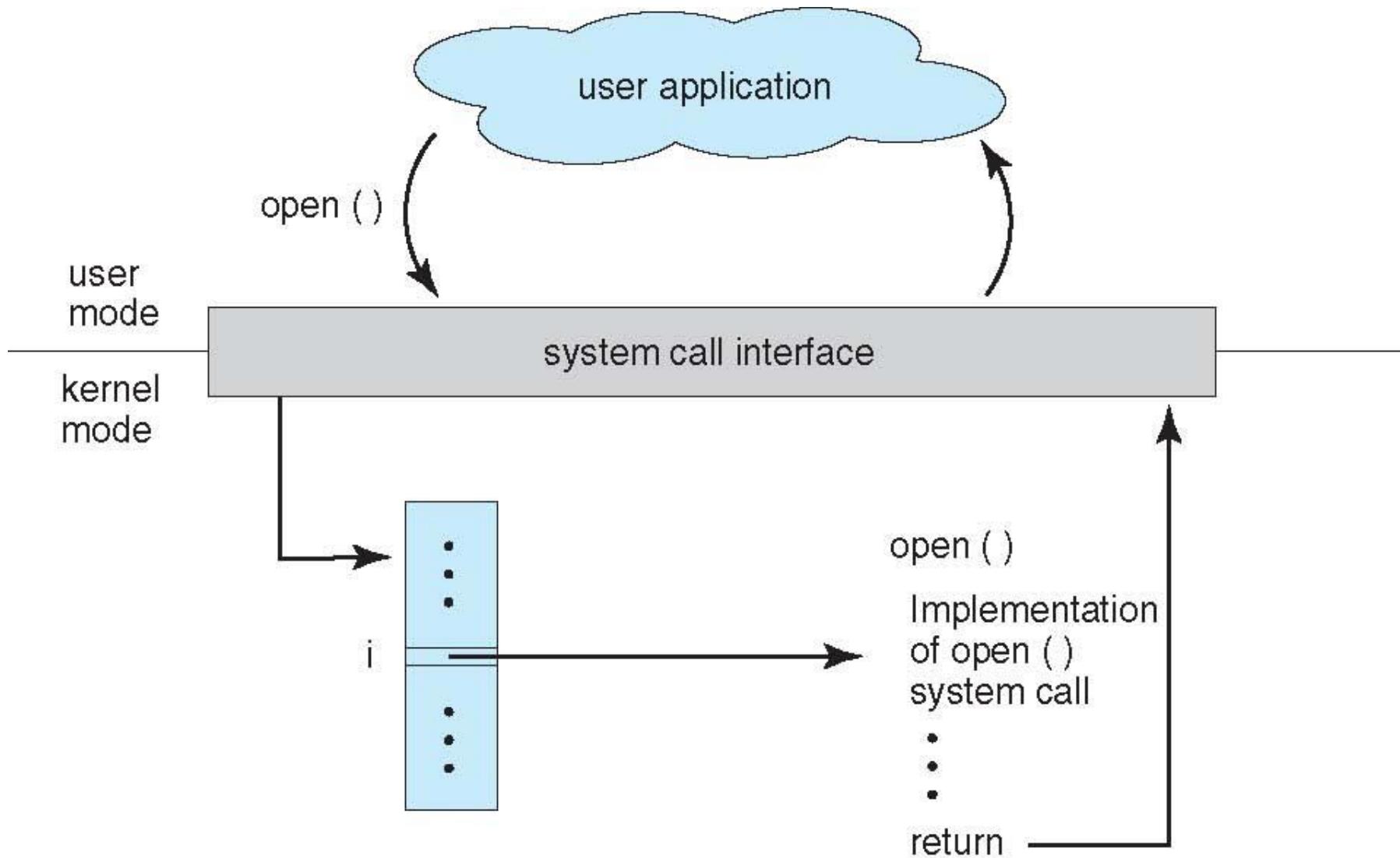
# System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)





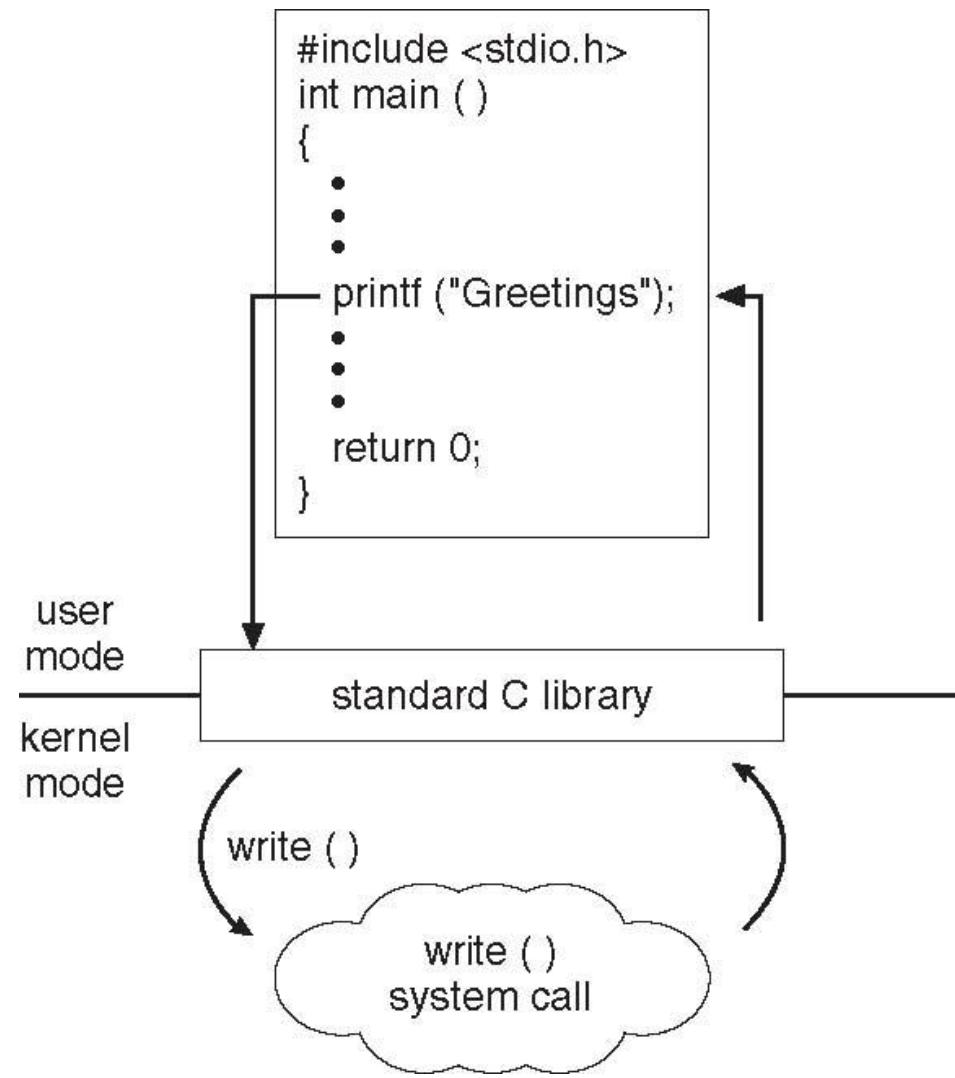
# API – System Call – OS Relationship





# Standard C Library Example

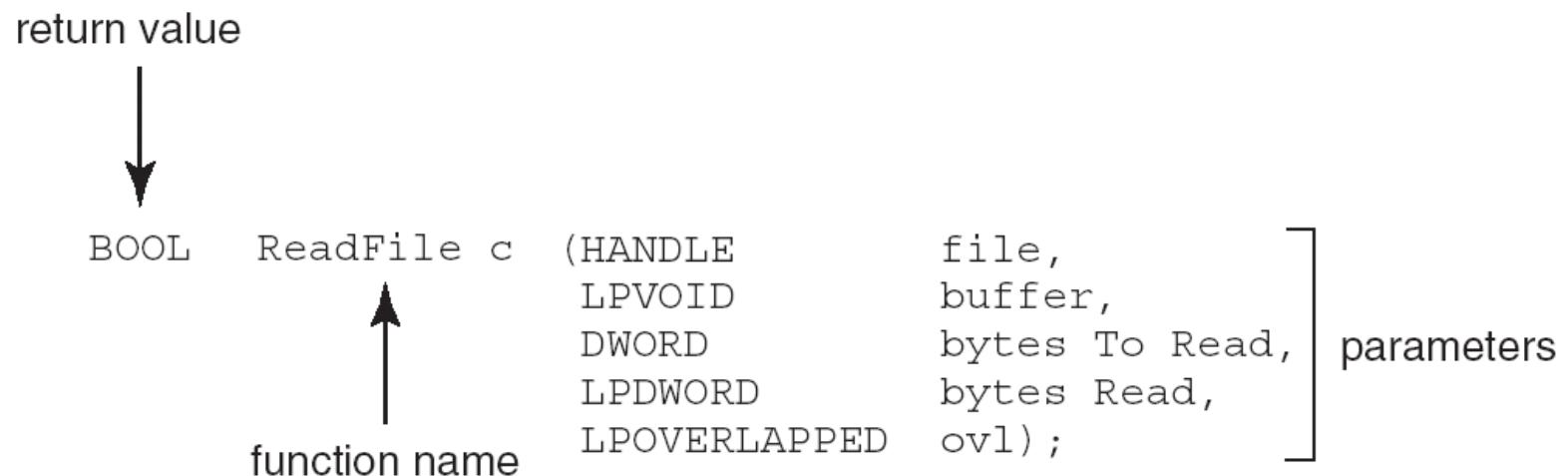
- C program invoking printf() library call, which calls write() system call





# Example of Standard API

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file



- A description of the parameters passed to `ReadFile()`
  - `HANDLE` `file`—the file to be read
  - `LPVOID` `buffer`—a buffer where the data will be read into and written from
  - `DWORD` `bytesToRead`—the number of bytes to be read into the buffer
  - `LPDWORD` `bytesRead`—the number of bytes read during the last read
  - `LPOVERLAPPED` `ovl`—indicates if overlapped I/O is being used





```
STRING DB 100 DUP(?)  
DATA    ENDS  
CODE SEGMENT
```

.... ....

```
MOV AX,DATA  
MOV DS,AX
```

.... ....

```
LEA   DX,MAXLEN  
MOV AH,10  
INT 21H
```

.... ....





# System Call Implementation

- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - ▶ Managed by run-time support library (set of functions built into libraries included with compiler)





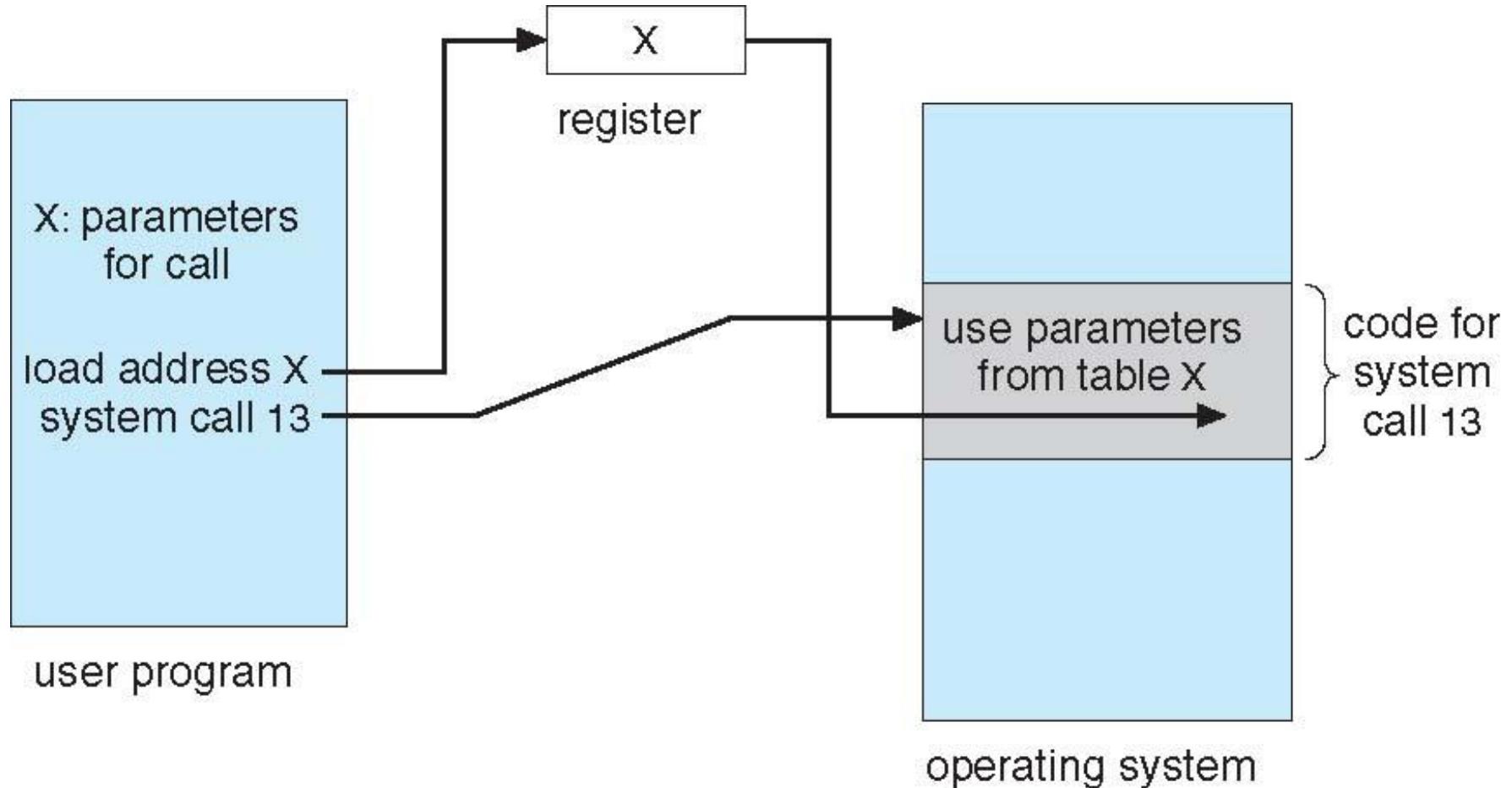
# System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - ▶ In some cases, may be more parameters than registers
  - Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
    - ▶ This approach taken by Linux and Solaris
  - Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed





# Parameter Passing via Table





# Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

C





# System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
  - File manipulation
  - Status information
  - File modification
  - Programming language support
  - Program loading and execution
  - Communications(browser, network sharing)
  - Application programs(calculator, notebook)
- Most users' view of the operation system is defined by system programs, not the actual system calls





# System Programs

- Provide a convenient environment for program development and execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex
- File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- Status information
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
  - Typically, these programs format and print the output to the terminal or other output devices
  - Some systems implement a registry - used to store and retrieve configuration information





# System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems
  - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another





# Simple Structure

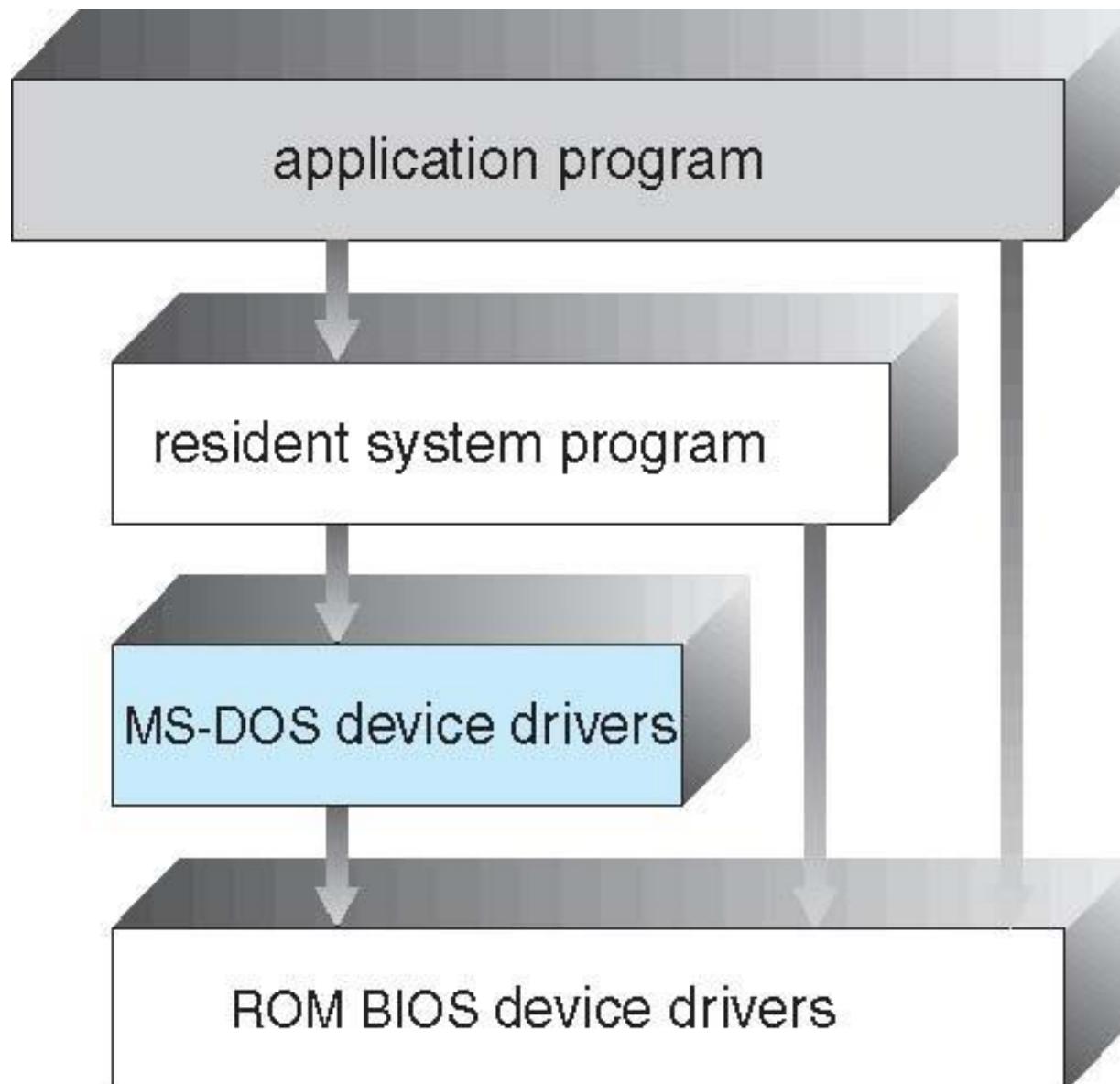
---

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



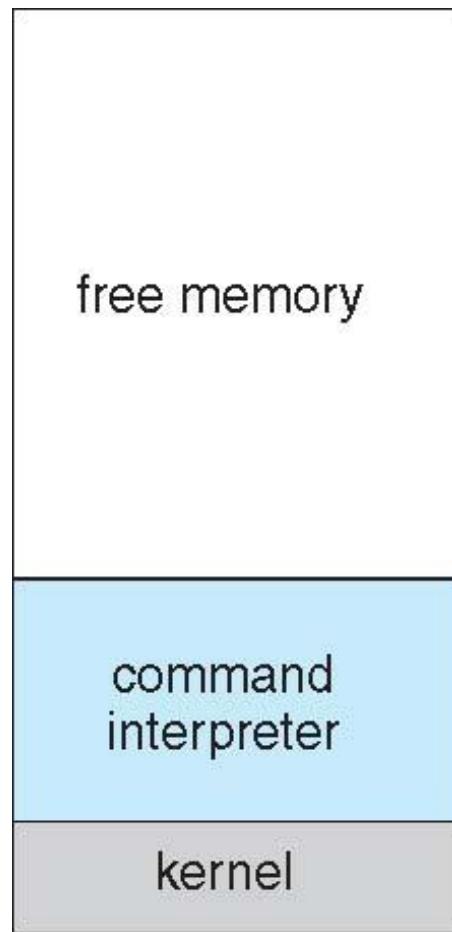


# MS-DOS Layer Structure

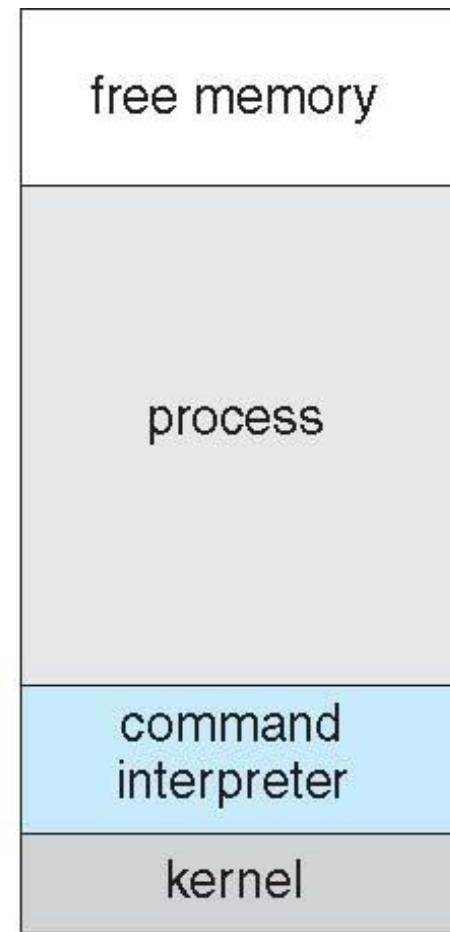




# MS-DOS execution



(a)



(b)

(a) At system startup (b) running a program





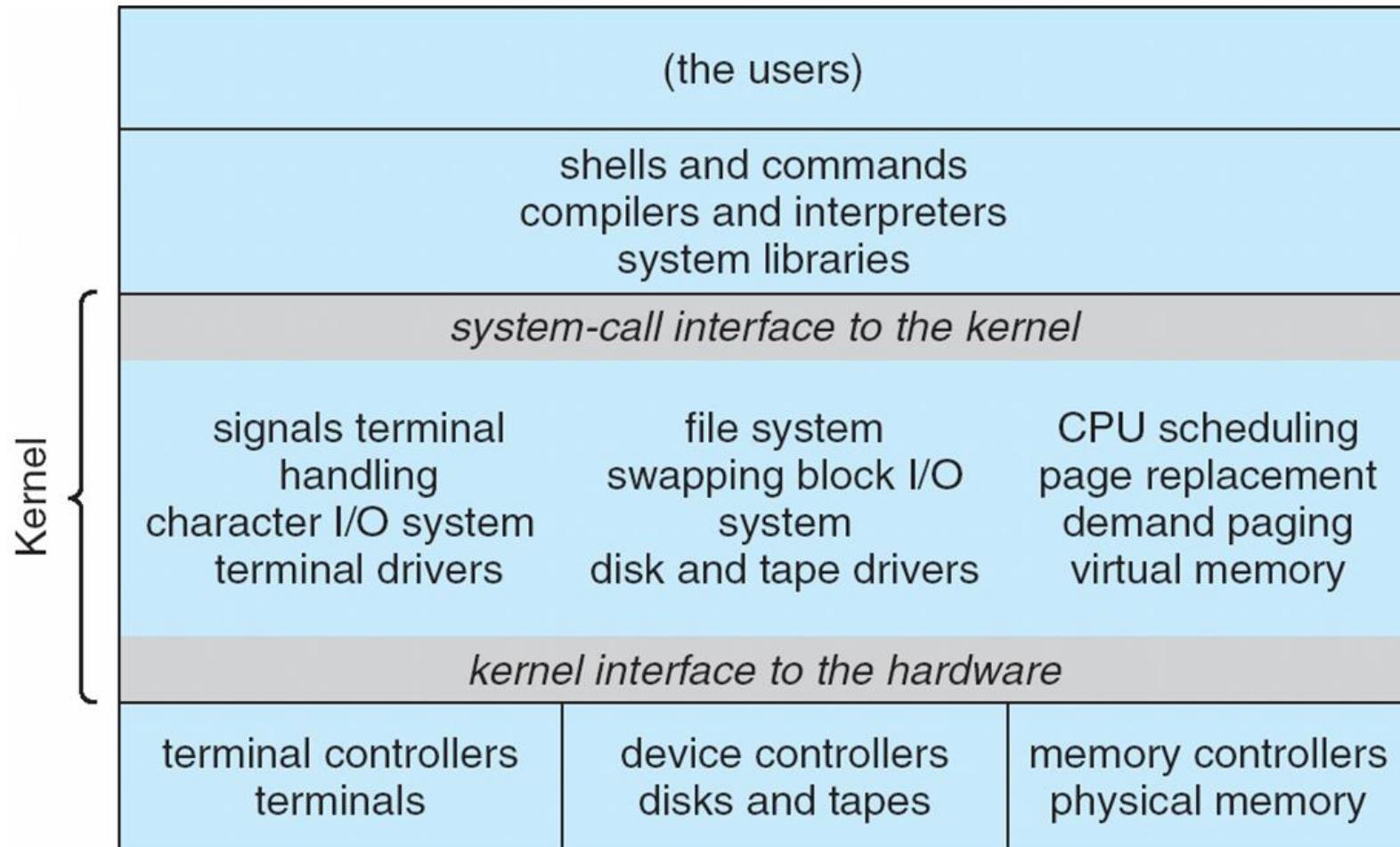
# Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



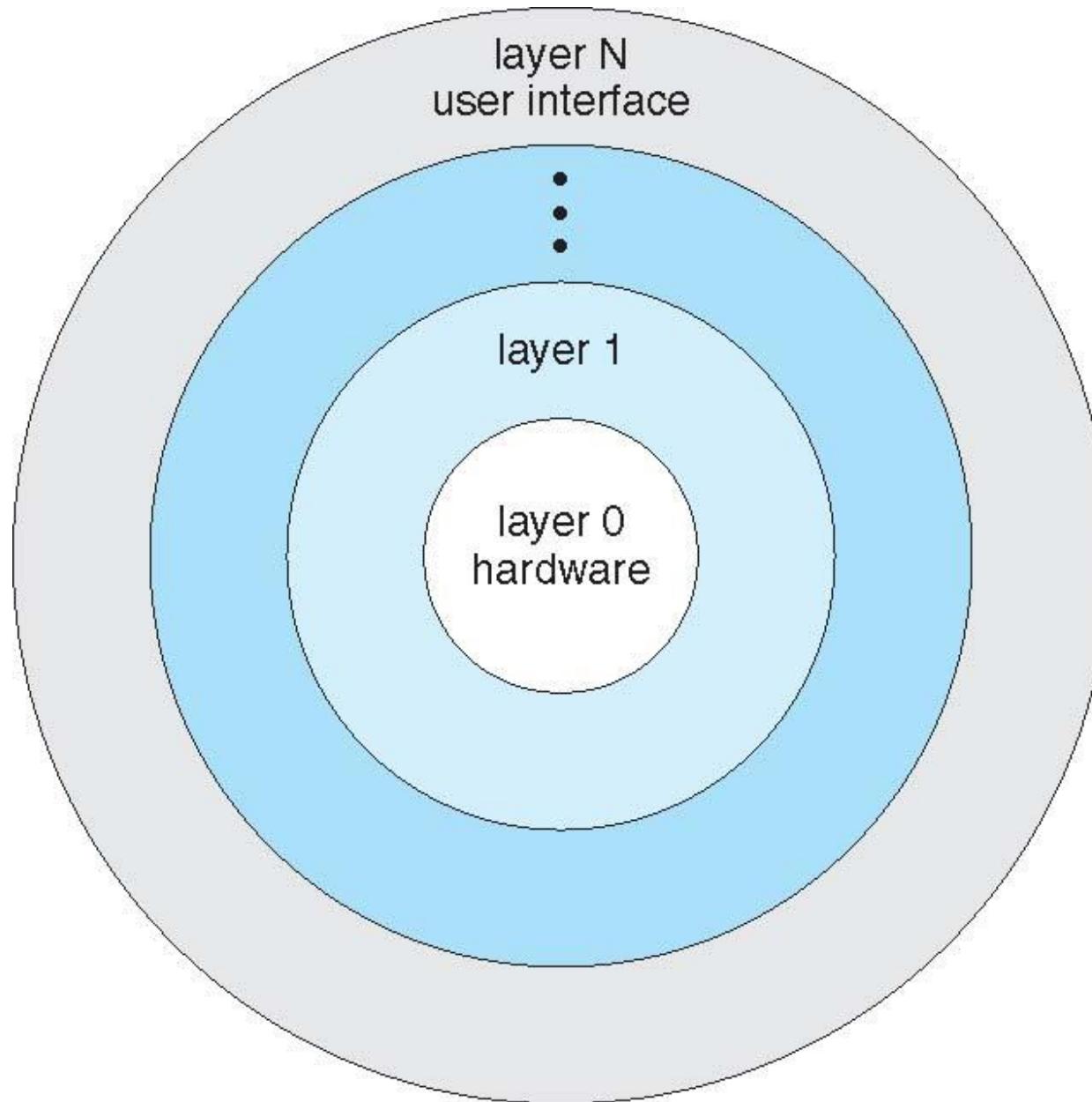


# Traditional UNIX System Structure





# Layered Operating System





# UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - ▶ Consists of everything below the system-call interface and above the physical hardware
    - ▶ Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level





# Microkernel System Structure

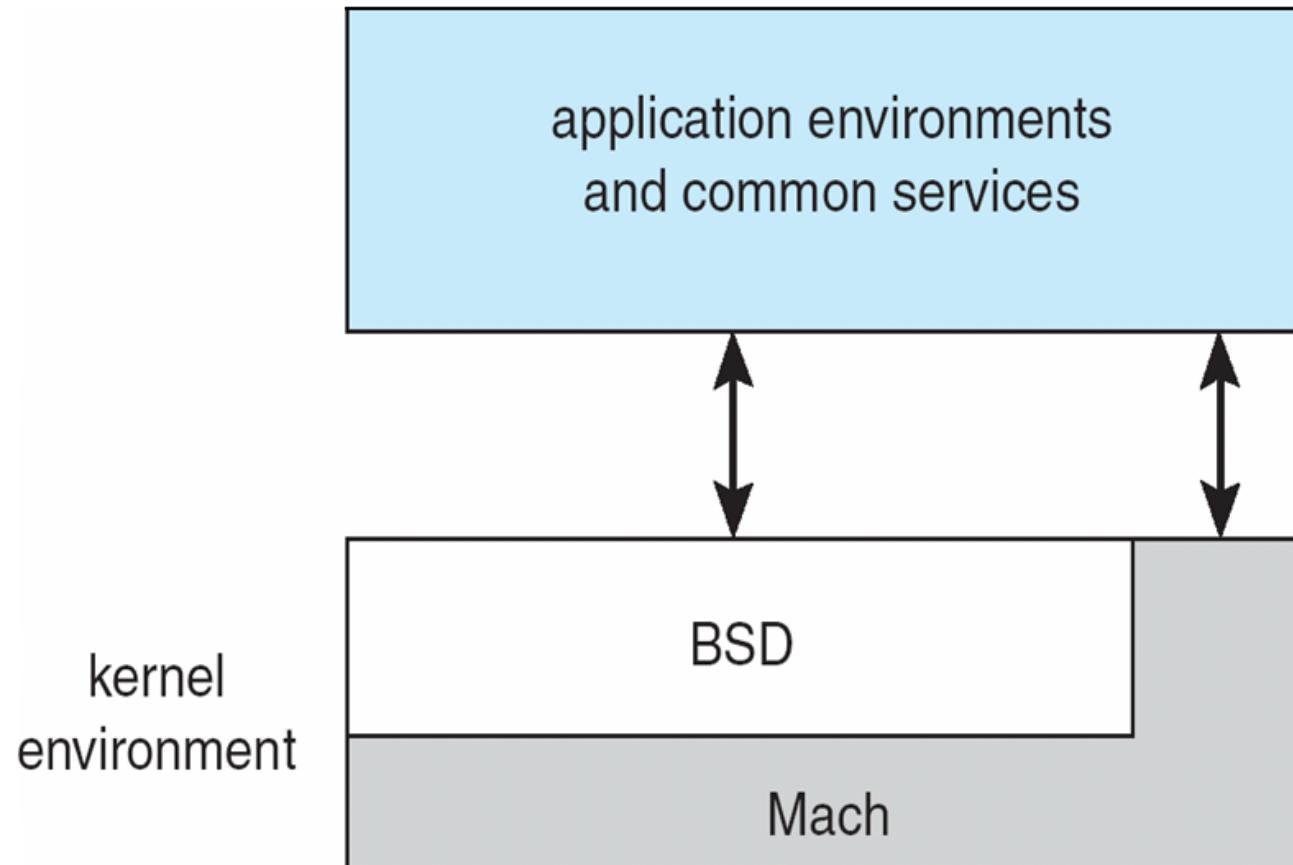
---

- Moves as much from the kernel into “*user*” space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication





# Mac OS X Structure





# Virtual Machines

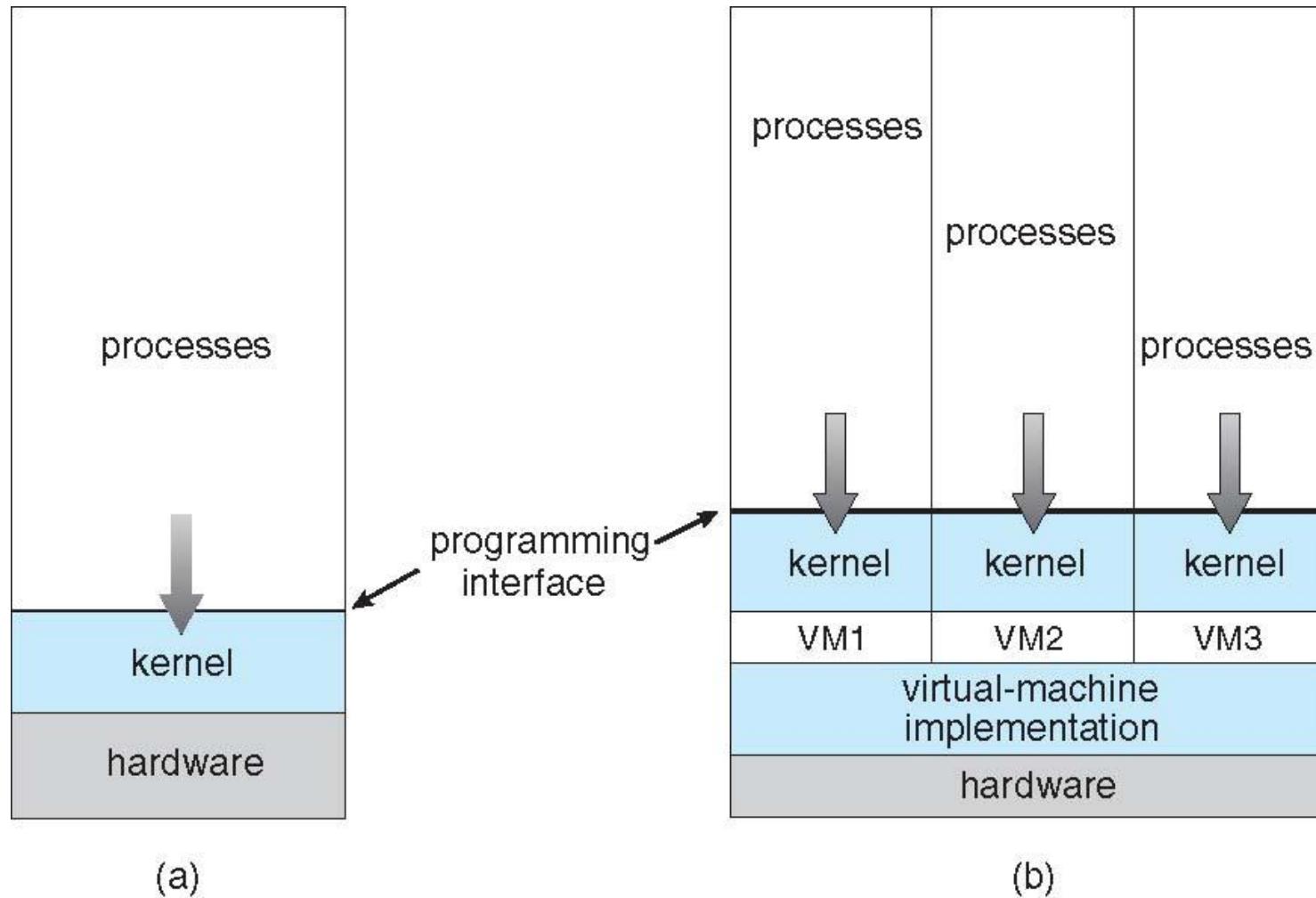
---

- A **virtual machine** takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system **host** creates the illusion that a process has its own processor and (virtual) memory.
- Each **guest** is provided with a (virtual) copy of underlying computer.





# Virtual Machines (Cont.)

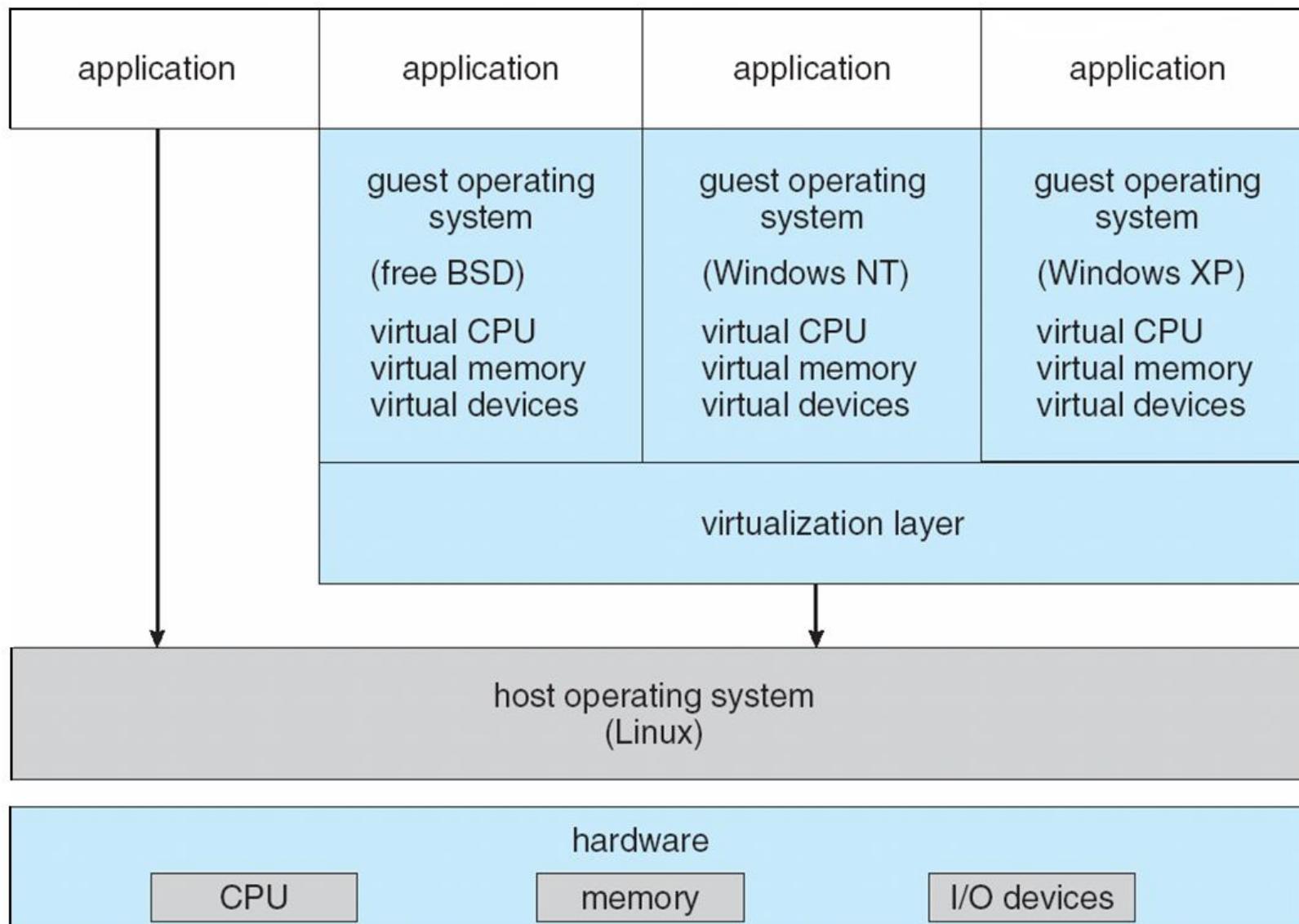


(a) Nonvirtual machine (b) virtual machine





# VMware Architecture





# Java

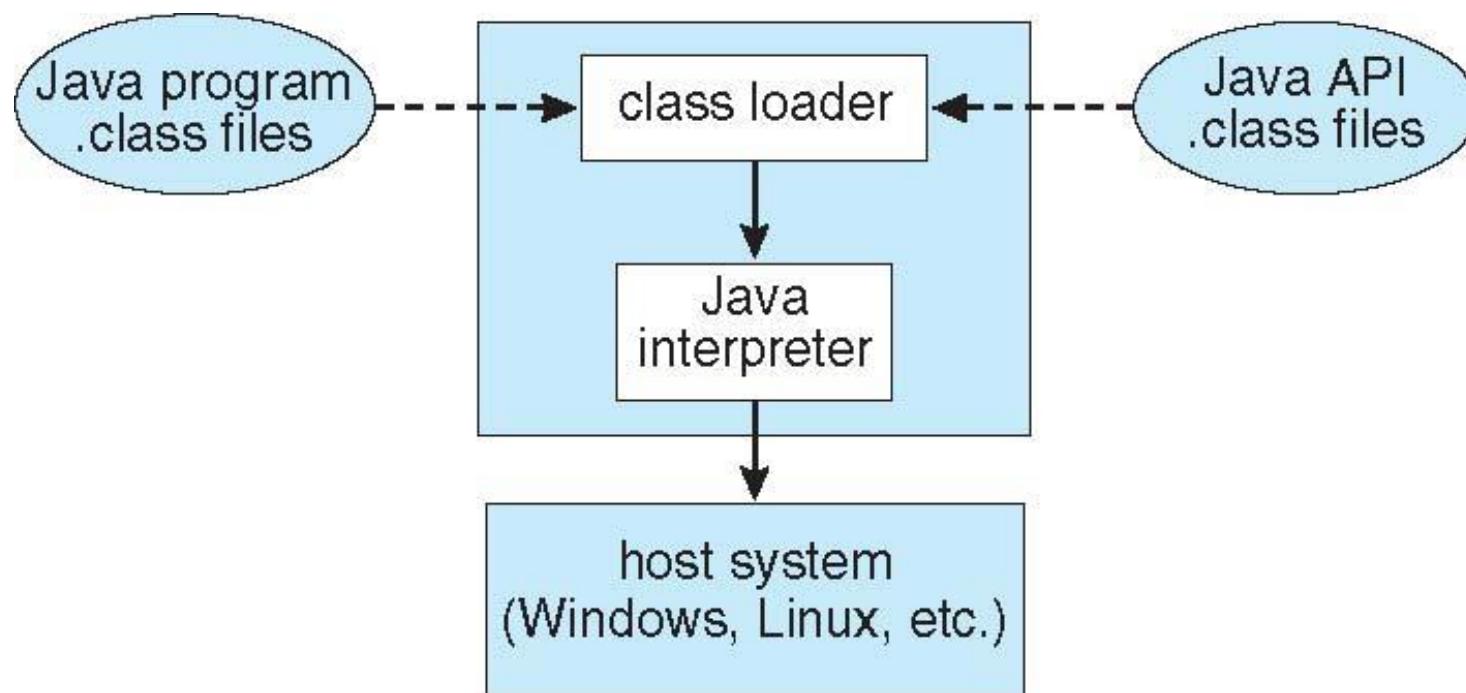
---

- Java consists of:
  1. Programming language specification
  2. Application programming interface (API)
  3. Virtual machine specification





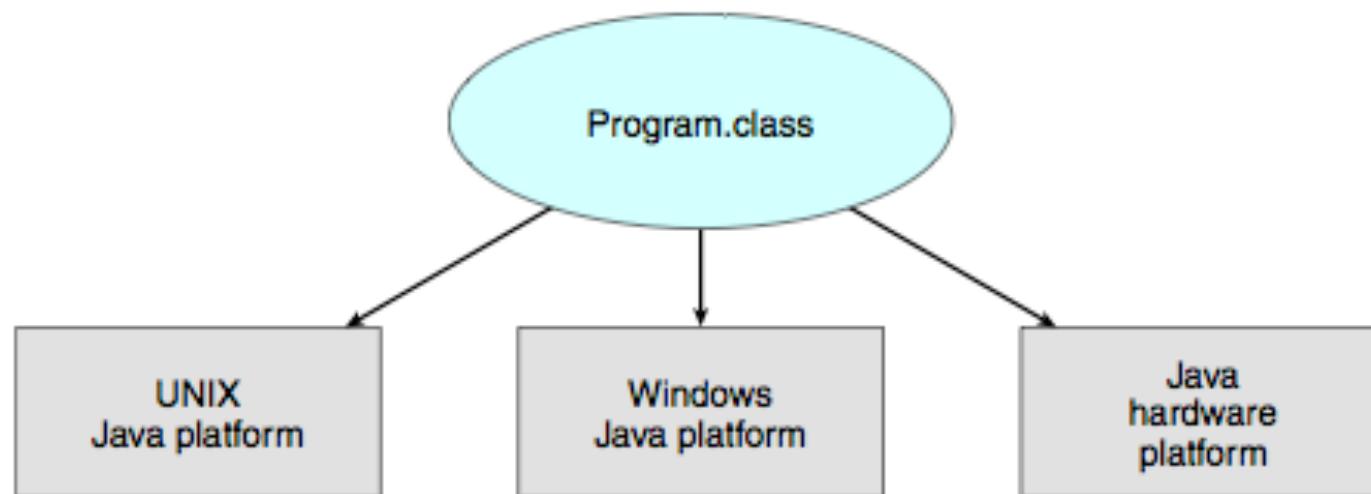
# The Java Virtual Machine





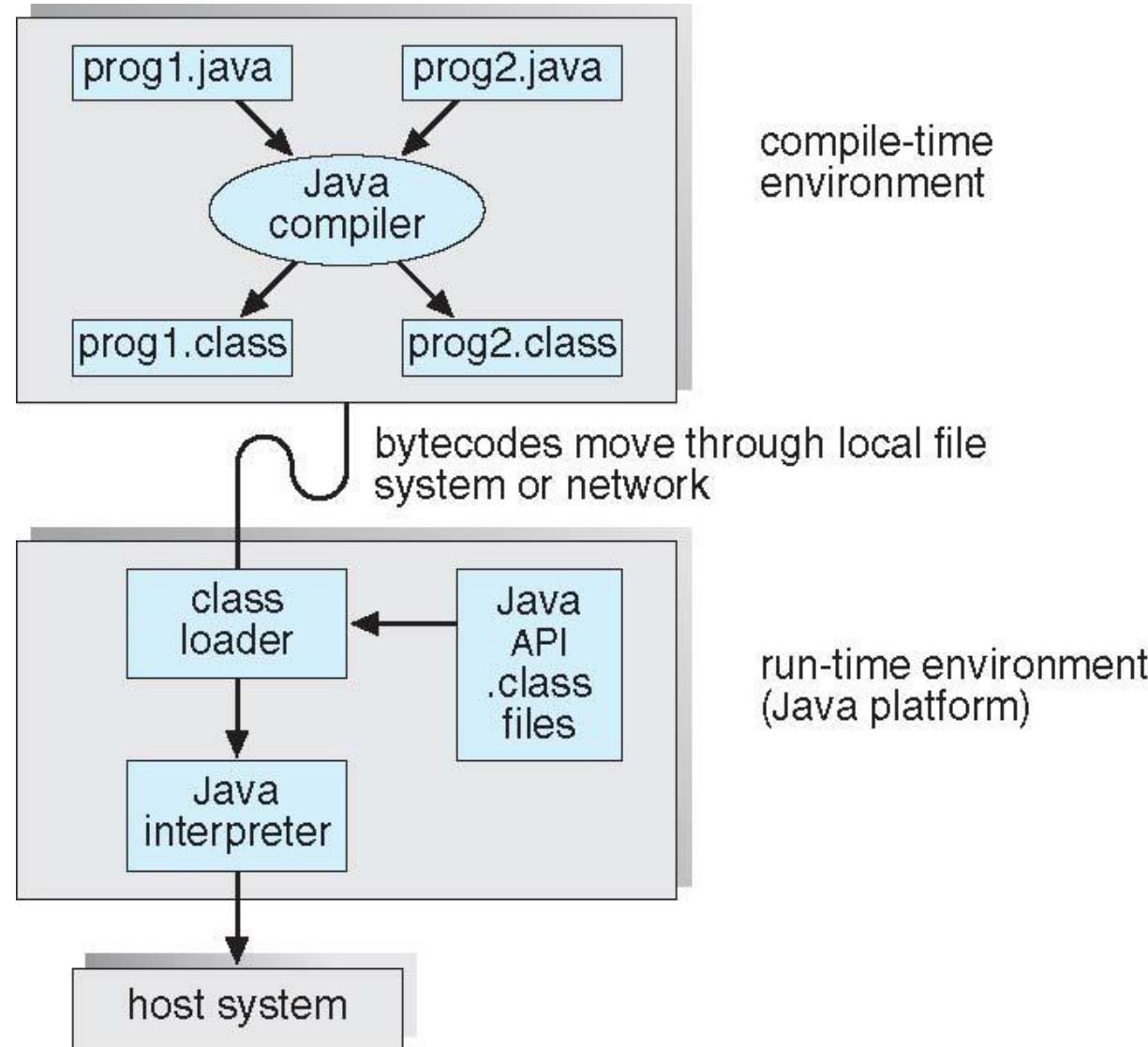
# The Java Virtual Machine

Java portability across platforms.





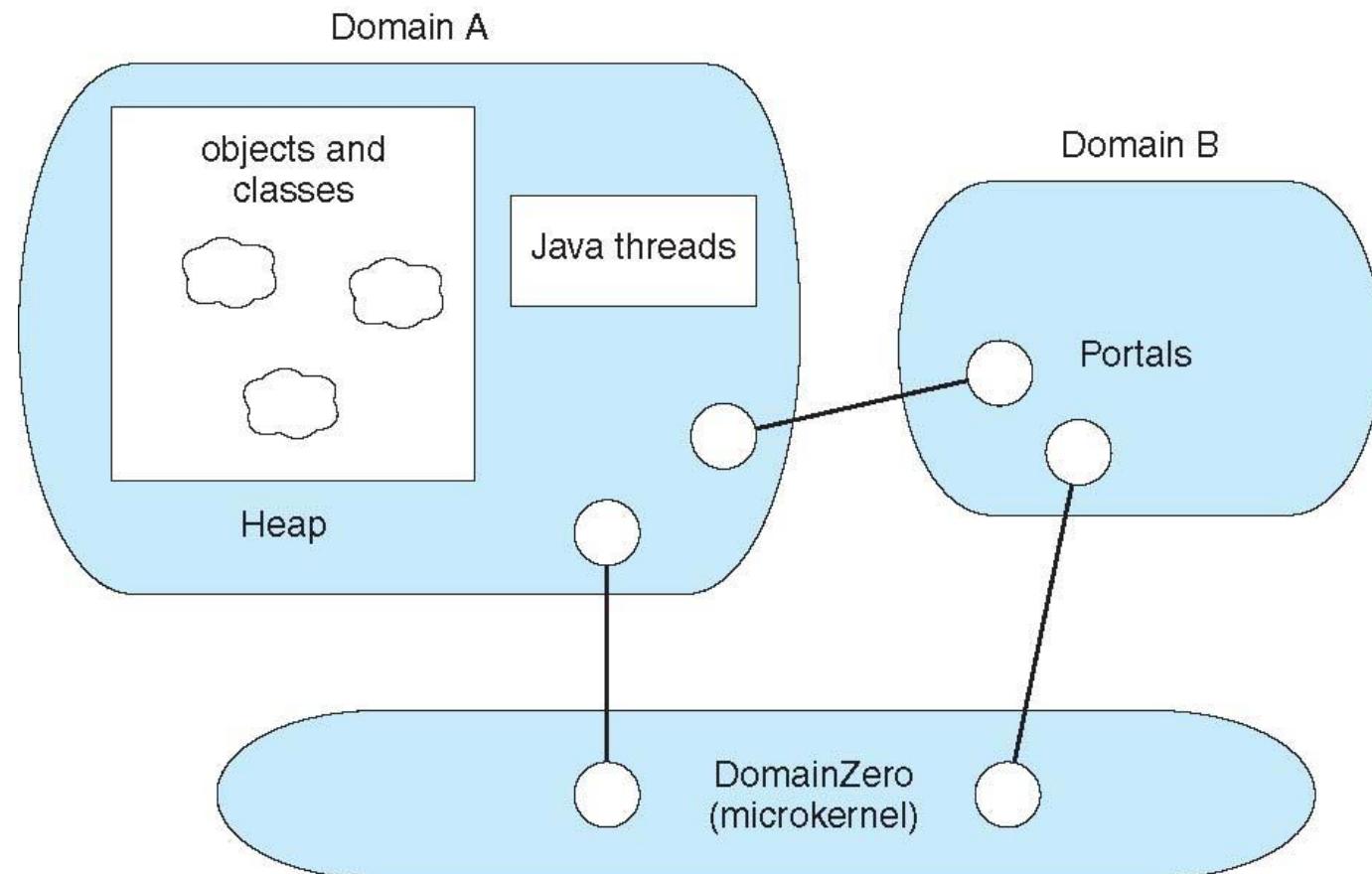
# The Java Development Kit





# Java Operating Systems

The JX operating system



Portals are the fundamental inter-domain communication mechanism





# Operating System Generation

---

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site.
- SYSGEN program obtains information concerning the specific configuration of the hardware system.
- *Booting* – starting a computer by loading the kernel.
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution.





# System Boot

- An operating system must be made available to hardware so hardware can start it.
  - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it.
  - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader.
  - When power initialized on system, execution starts at a fixed memory location.
    - ▶ Firmware is used to hold initial boot code.



# End of Chapter 2



# 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 describe communication in client-server systems





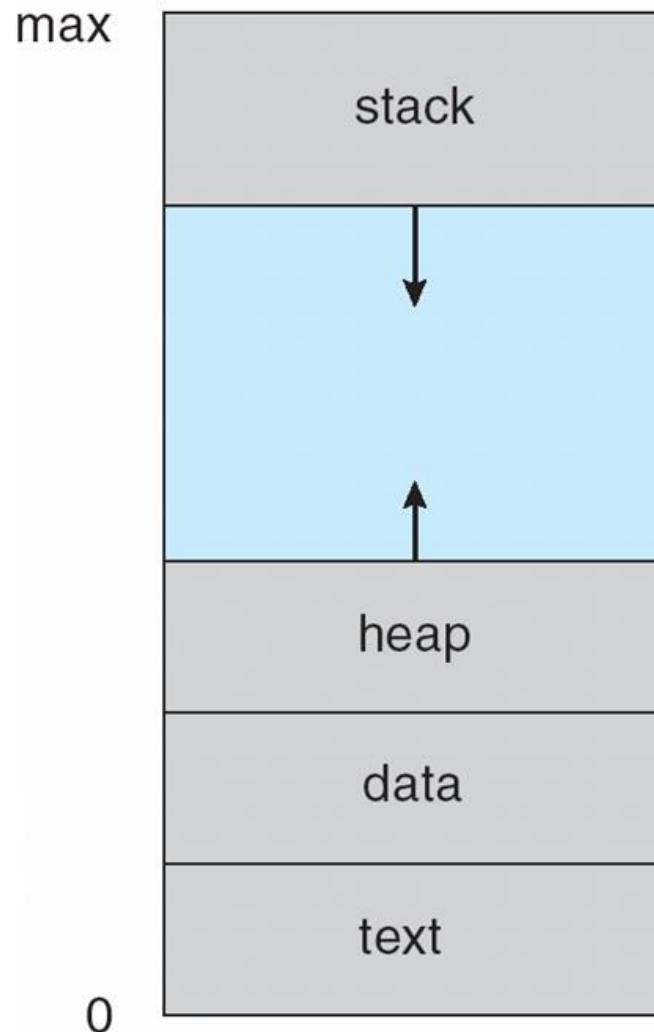
# Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs 作业
  - Time-shared systems – user programs or tasks 任务
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a **program in execution**; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section





# 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

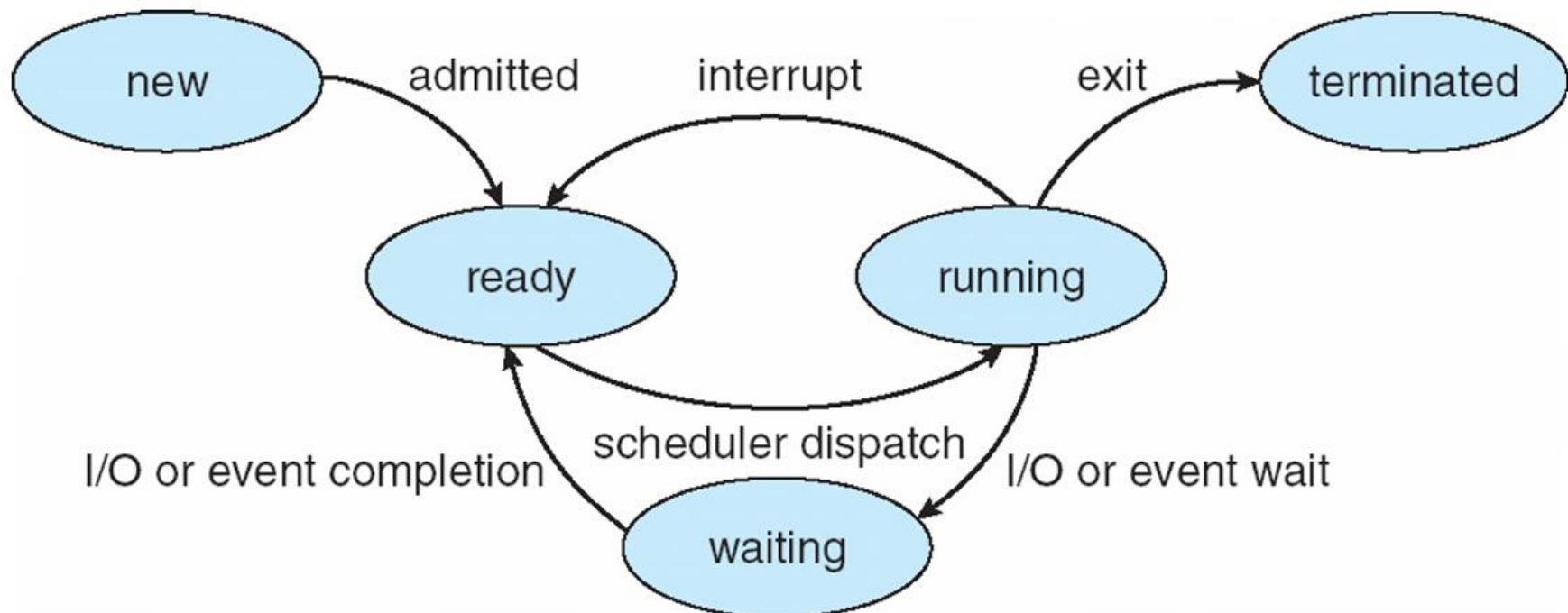




# Diagram of Process State

图应该要会画会描述

Interrupt: time slice run out





# Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



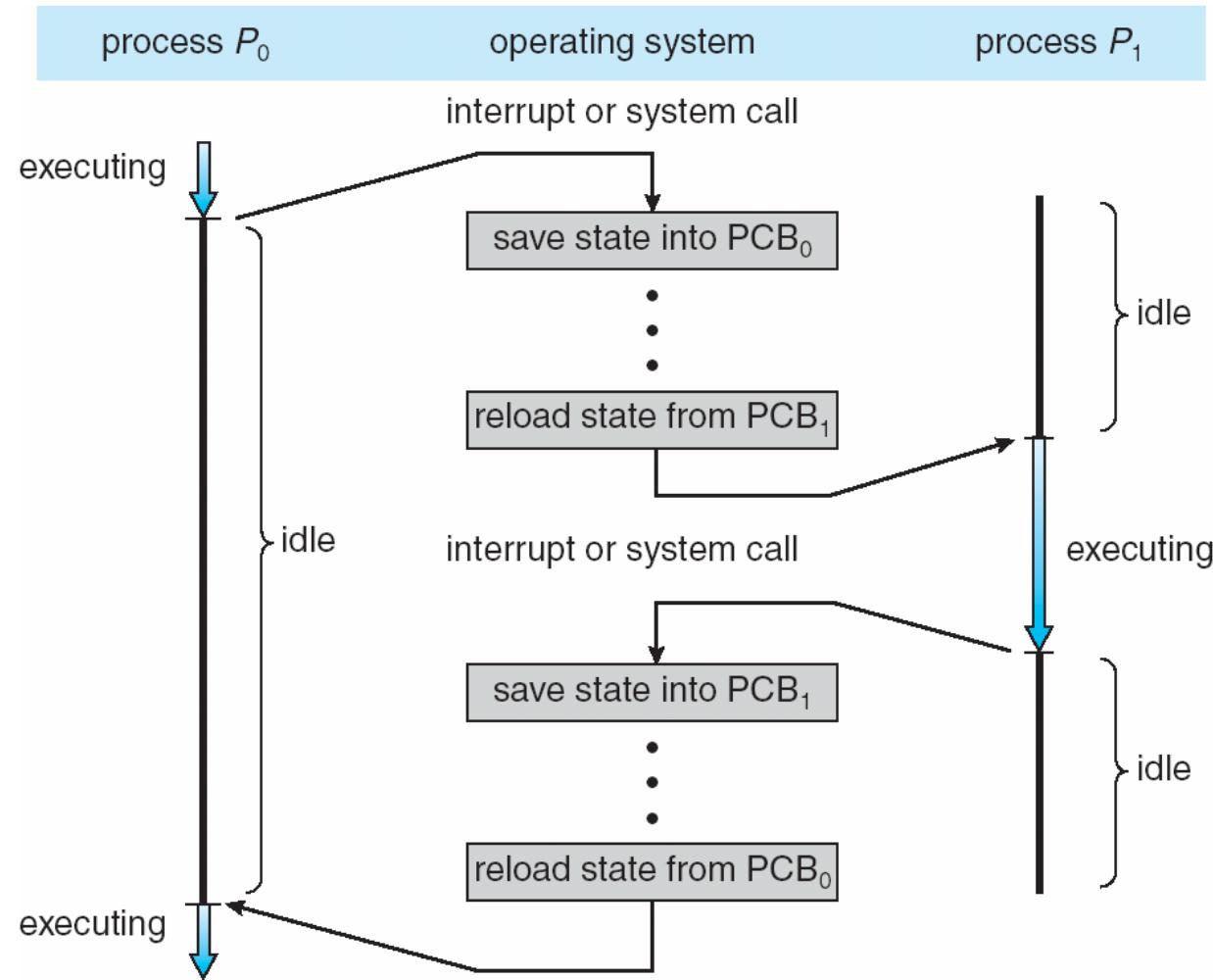


# Process Control Block (PCB)





# CPU Switch From Process to Process





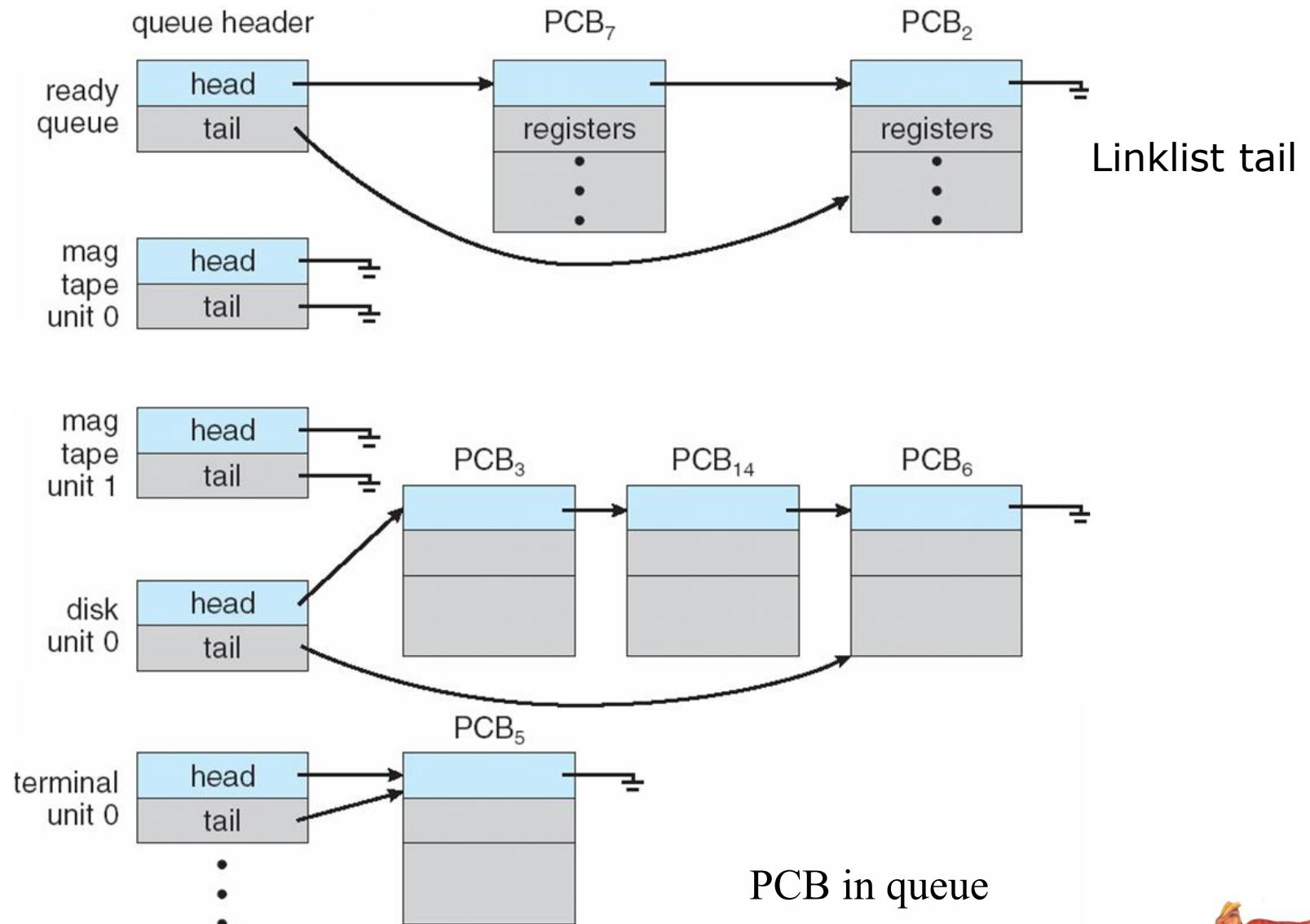
# Process Scheduling Queues

- **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



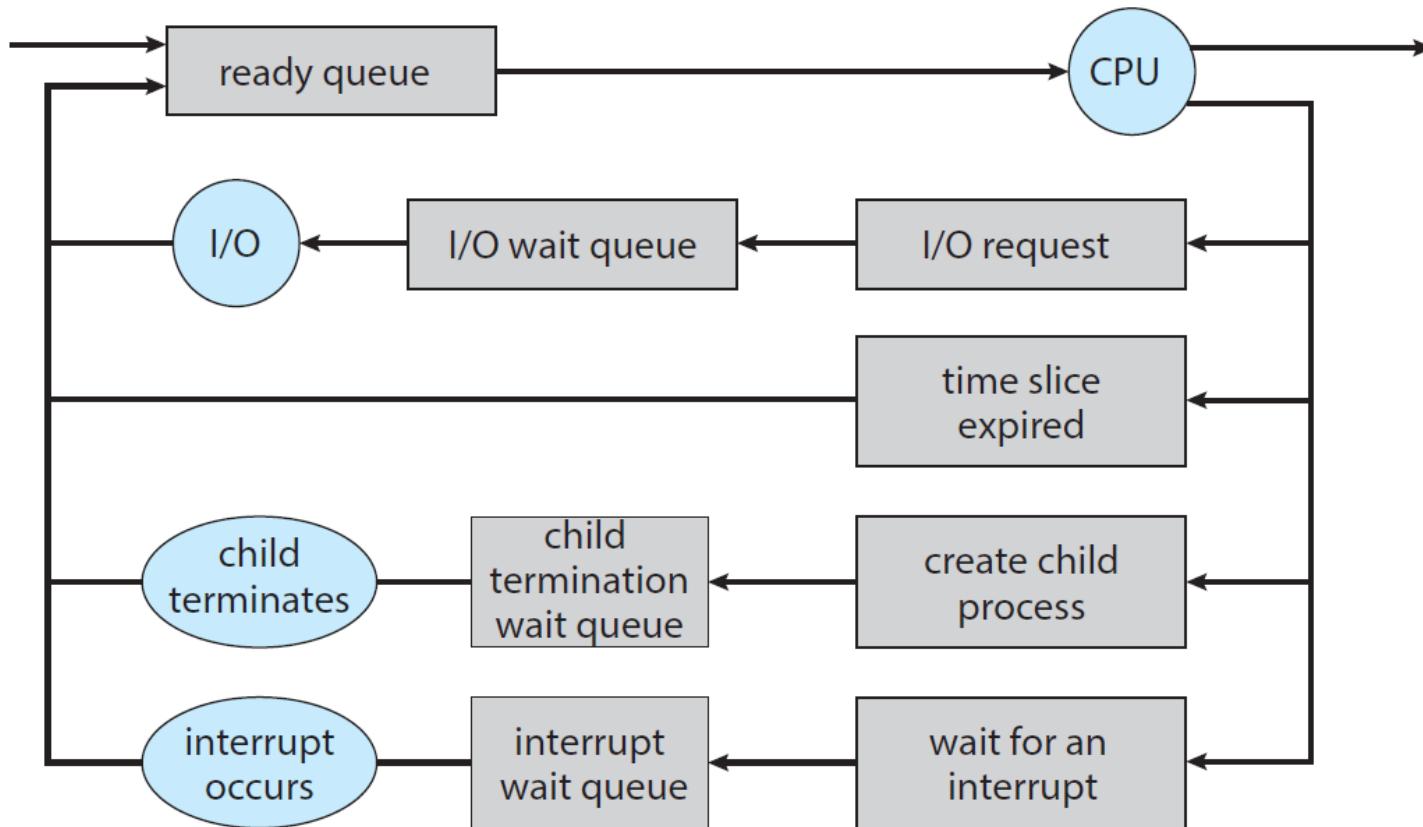


# Ready Queue And Various I/O Device Queues





# Representation of Process Scheduling



**Figure 3.5** Queueing-diagram representation of process scheduling.





- The process could issue an I/O request and then be placed in an I/O wait queue.
- The process could create a new child process and then be placed in a wait queue while it awaits the child's termination.(父进程进入等待队列)
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue.  
interrupt refers to current program is interrupted by high level interrupt





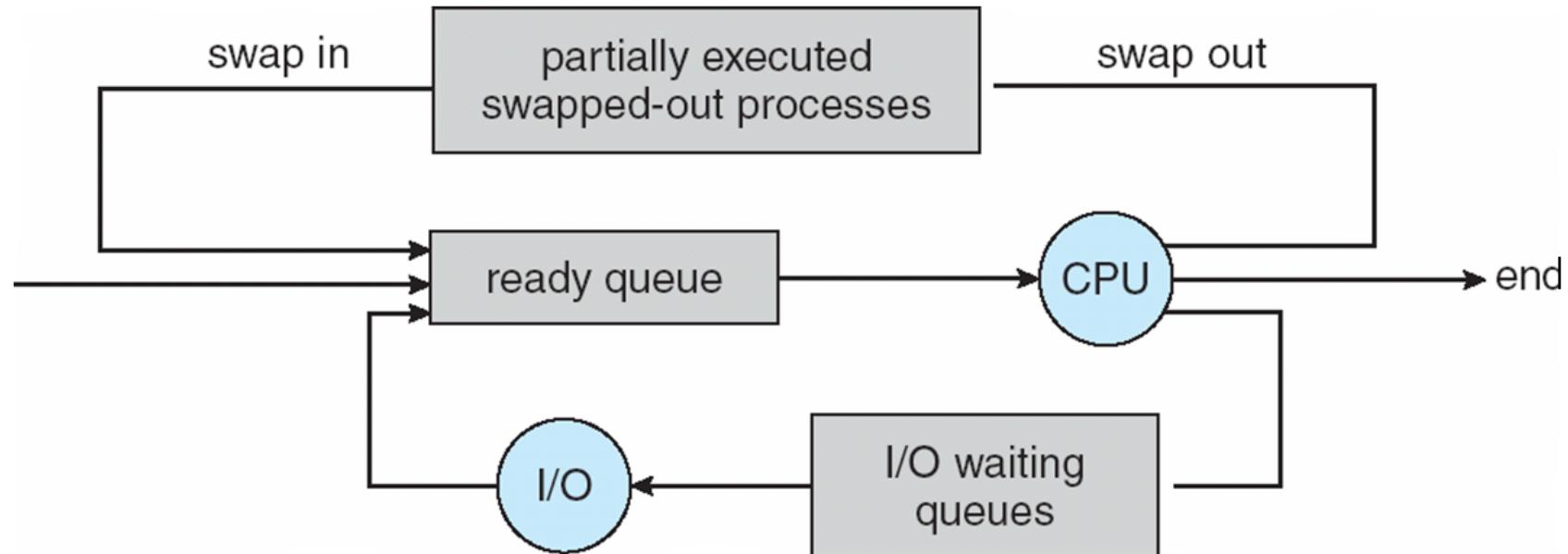
# Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
- **medium-term scheduler** (or CPU scheduler) – selects which process should be swapped out and swapped in memory





# Addition of Medium Term Scheduling



“swap out” from memory to disk, where its current status is saved

“swapped in” from disk back to memory, where its status is restored.

only necessary when memory has been overcommitted and must be freed up.





# Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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 I/O密集型
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts CPU密集型





# 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**(上下文)
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support





# 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
  - Children share subset of parent's resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate





# Process Creation (Cont.)

---

## ■ Address space

- Child duplicate of parent  
(copy, same physical mem, different logical mem)
- Child has a program loaded into it

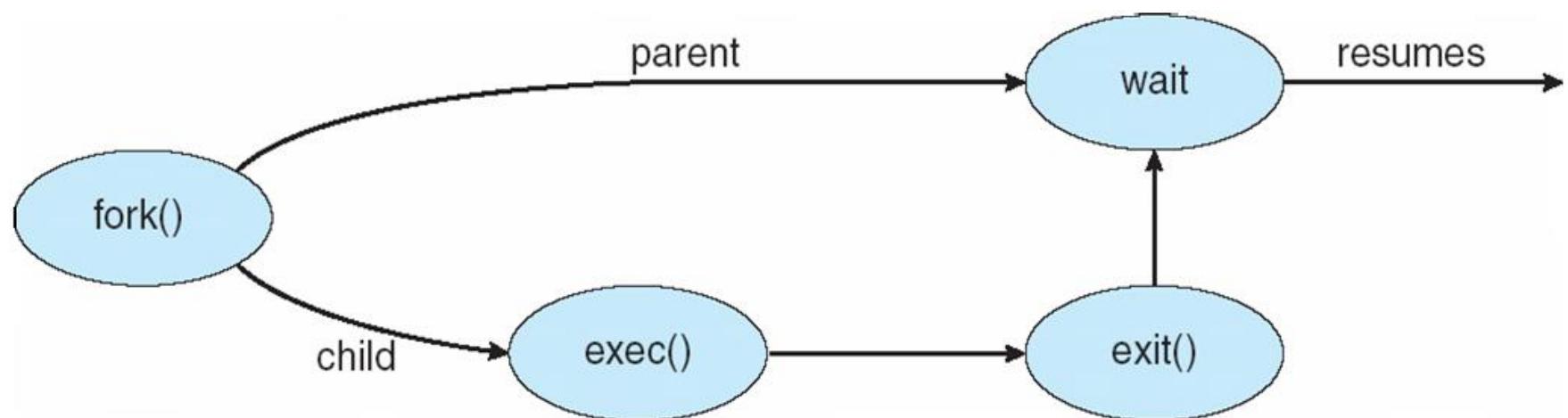
## ■ UNIX examples

- **fork** system call creates new process
- **exec** system call used after a **fork** to replace the process' memory space with a new program





# Process Creation





# C Program Forking Separate Process

```
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-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");
        exit(0);
    }
}
```





# Process Creation in POSIX

```
#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");
        exit(-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");
        exit(0);
    }
}
```

Navigation icons: back, forward, search, etc.





# Process Creation in Win32

```
#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 line
    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 Creation in Java

```
import java.io.*;

public class OSProcess
{
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Usage: java OSProcess <command>");
            System.exit(0);
        }

        // args[0] is the command
        ProcessBuilder pb = new ProcessBuilder(args[0]);
        Process proc = pb.start();

        // obtain the input stream
        InputStream is = proc.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);

        // read what is returned by the command
        String line;
        while ( (line = br.readLine()) != null)
            System.out.println(line);

        br.close();
    }
}
```



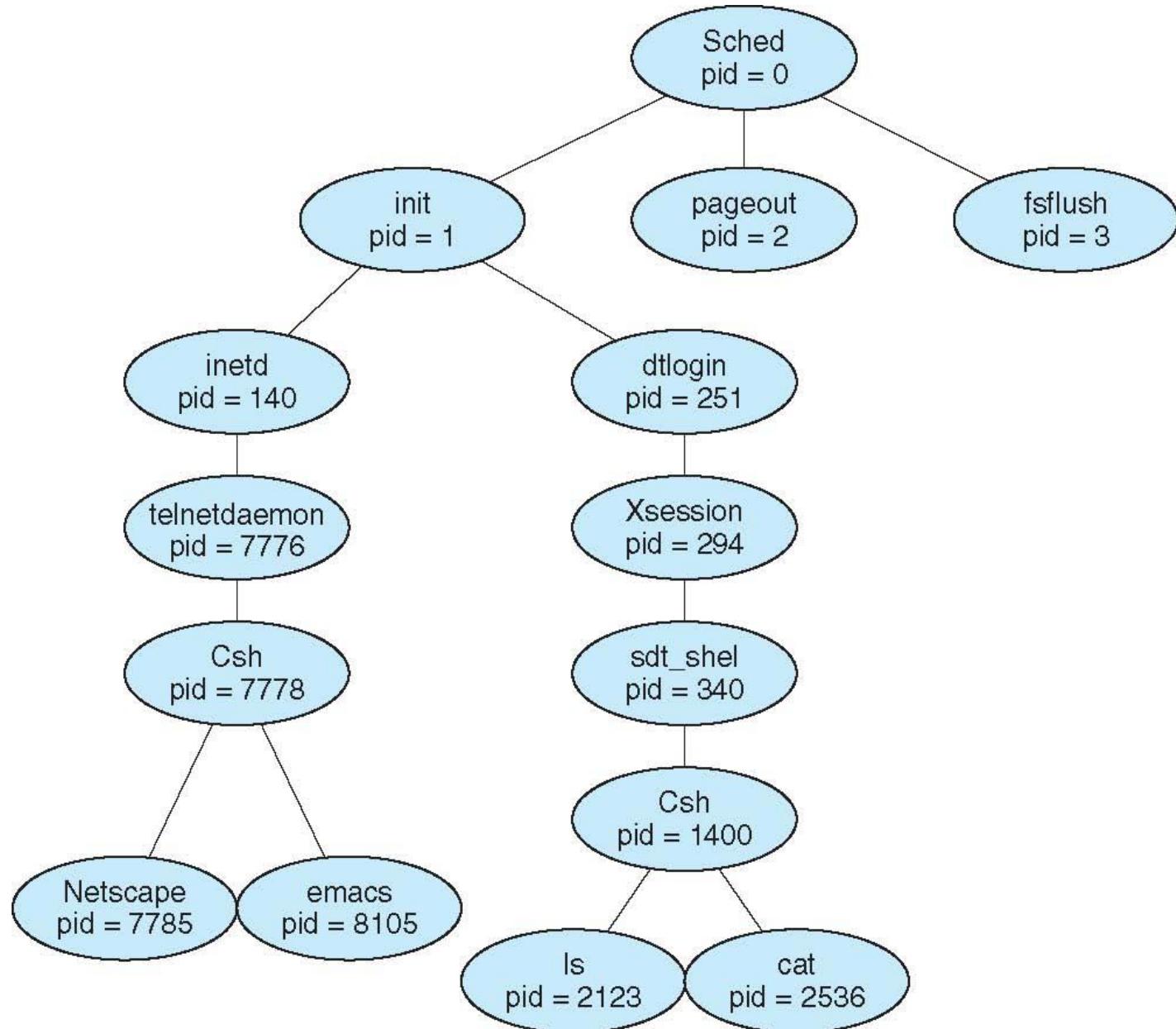


- Class Process method `getInputStream()`
- Get child process input stream, which can receive data from child process
  - `InputStreamReader`:
  - Convert byte to char
  - `Bufferreader`





# A tree of processes on a typical Solaris





# Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - ▶ Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**
      - 
      - 级联终止





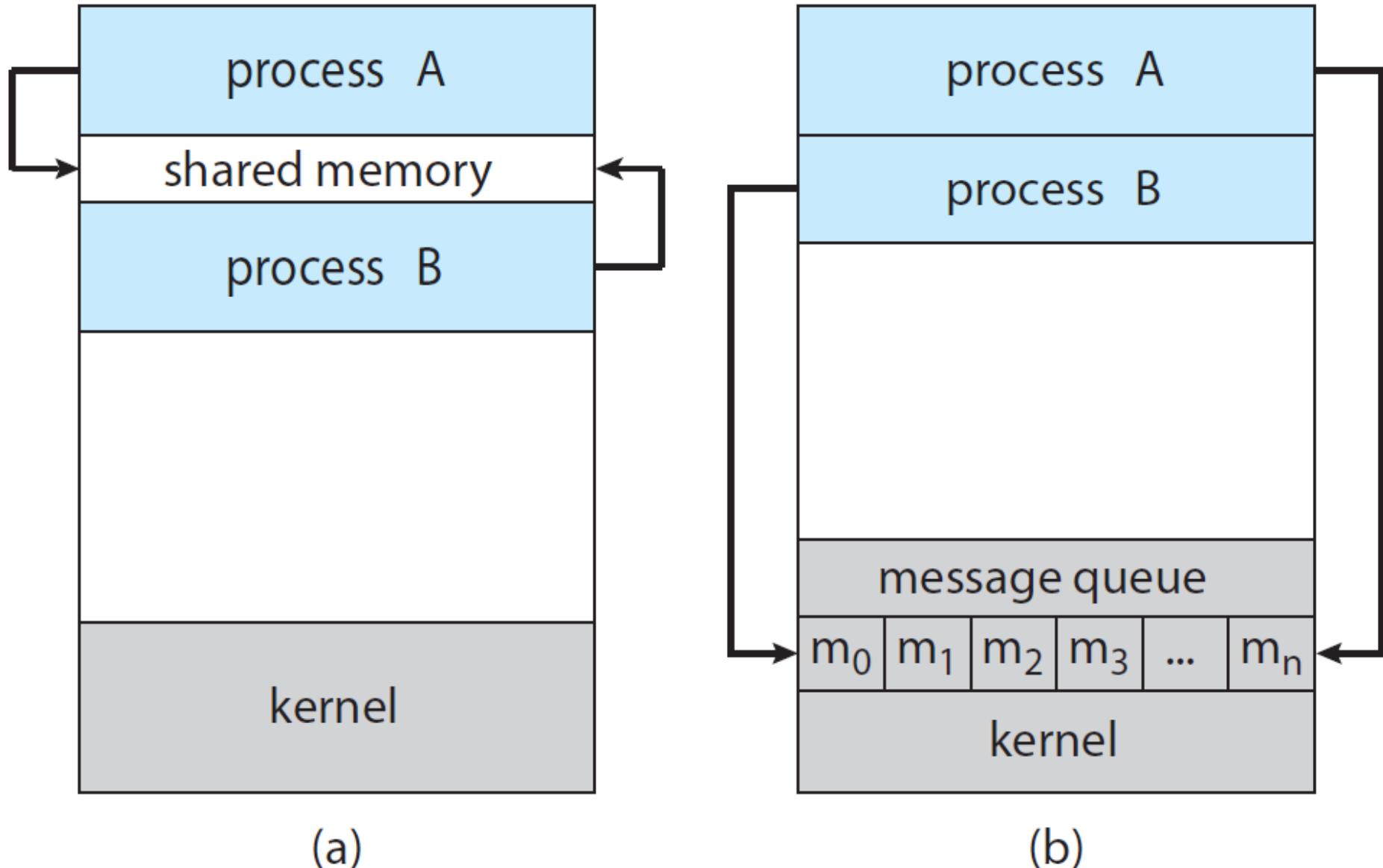
# Interprocess Communication

- 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





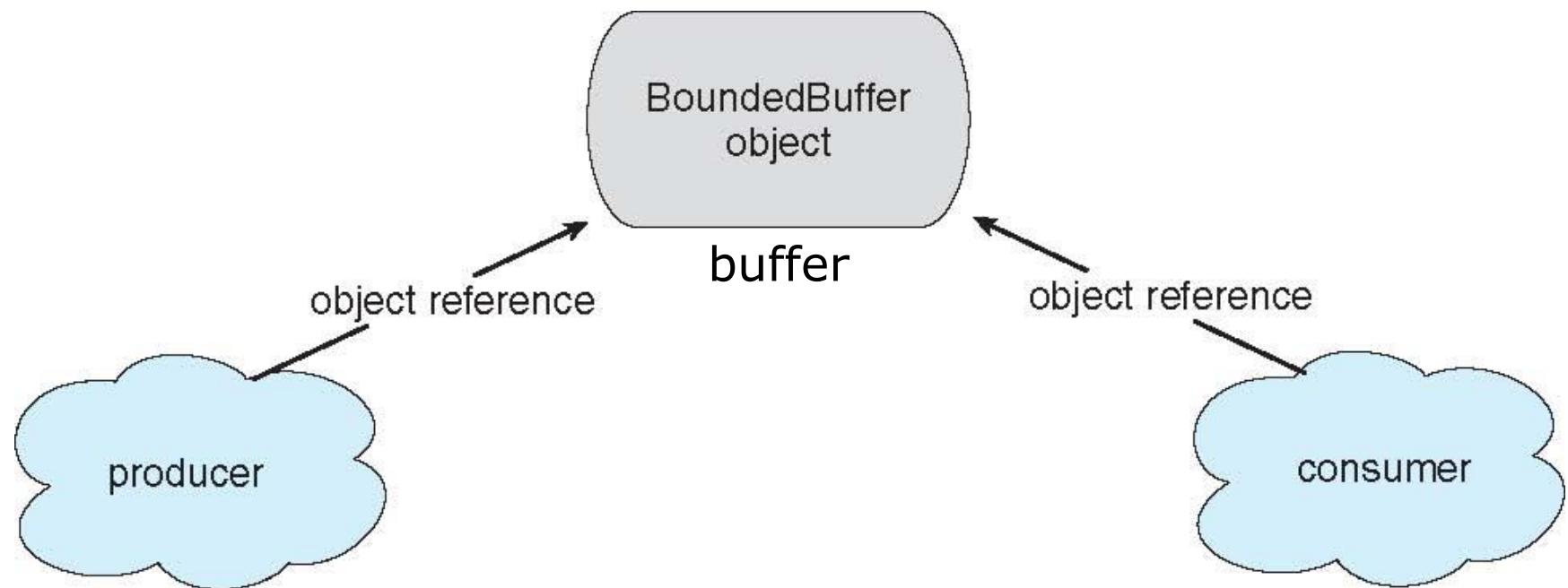
# Producer-Consumer Problem

- 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  
无界
  - *bounded-buffer* assumes that there is a fixed buffer size  
有界





# Simulating Shared Memory in Java





# Bounded-Buffer – Shared-Memory Solution

- Shared data in C

```
#define BUFFER_SIZE 10  
  
typedef struct {  
  
    ...  
  
} item;    //define item structure  
  
item buffer[BUFFER_SIZE];  
  
int in = 0; // pointer for writing  
  
int out = 0;// pointer for reading
```

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





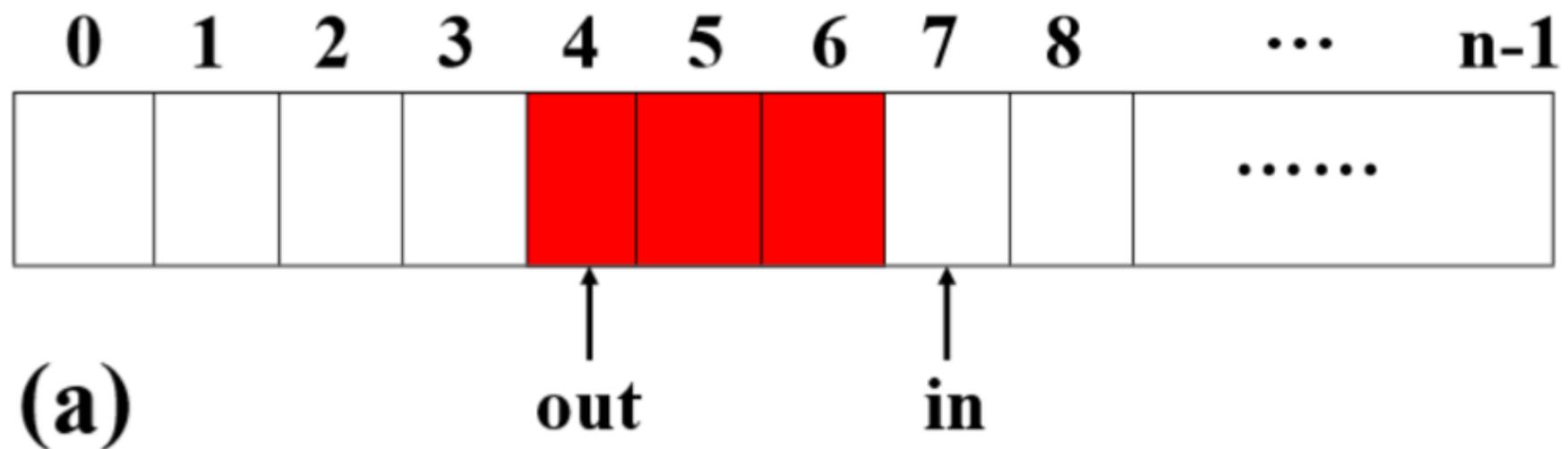
# Bounded-Buffer – Producer

```
while (true) {  
    /* Produce an item */  
  
    while (((in = (in + 1) % BUFFER SIZE ) == out)  
        ; /* do nothing -- no free buffers */ wait  
  
    buffer[in] = item;  
  
    in = (in + 1) % BUFFER SIZE;  
}
```

In is the variable in points to the next free position in the buffer;  
Out points to the first full position in the buffer.

% : for loop  
in, out are from zero







# Bounded Buffer – Consumer

---

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume, wait  
        //all writing has been read out→empty  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```





# 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
- Two operations:
  - **send(message)** – message size fixed or variable
  - **receive(message)**
- If  $P$  and  $Q$  wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)





# 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
- Local Procedure call ( C/S mode)





# 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
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.





# Indirect Communication

---

## ■ Operations

- create a new mailbox
- 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





# Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null





# Buffering

---

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages  
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





# Communications in Client-Server Systems

---

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





# Sockets

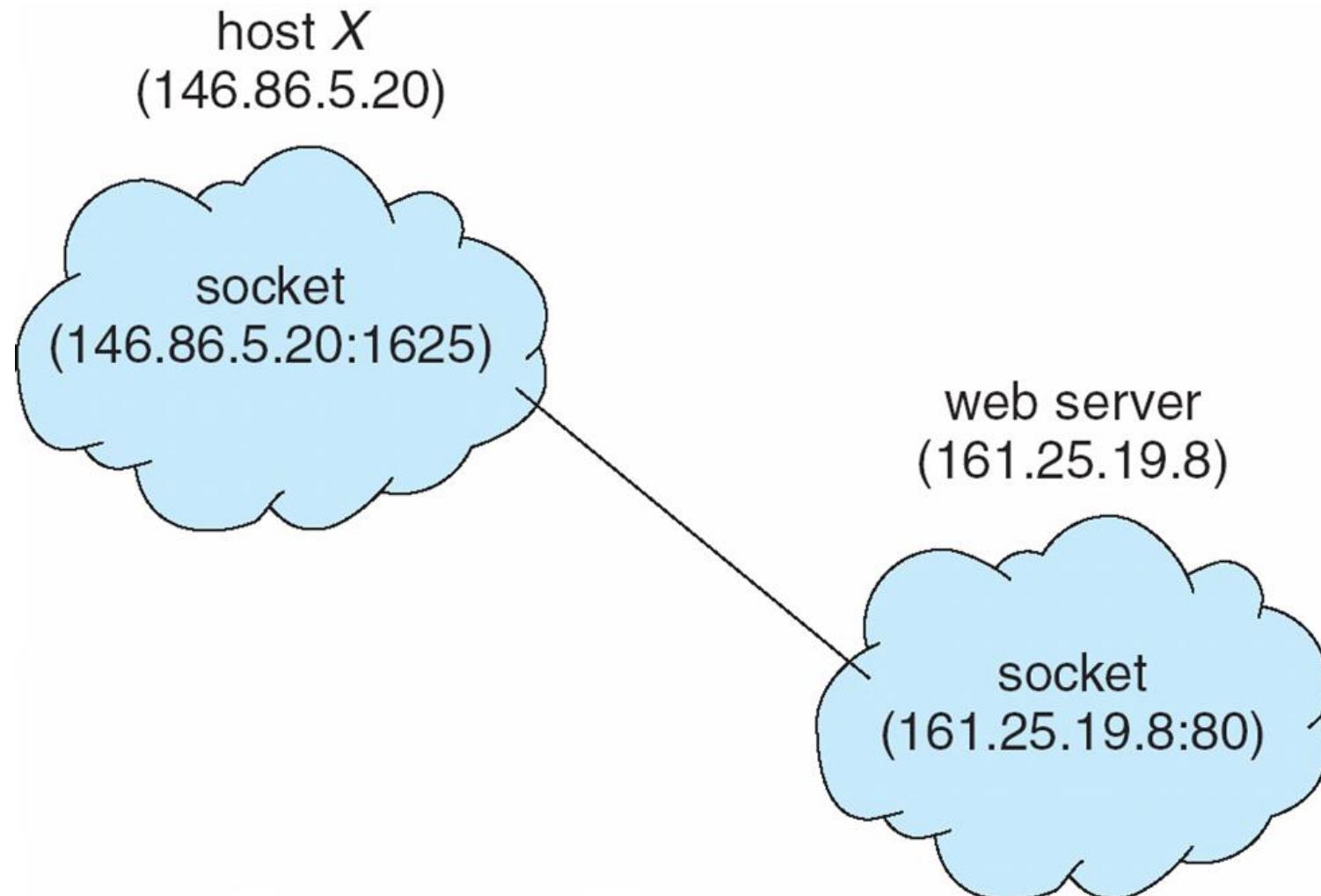
---

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets





# Socket Communication





# Socket Communication in Java

```
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);
        }
    }
}
```

Server write date





# Socket Communication in Java

```
public class DateClient
{
    public static void main(String[] args) {
        try {
            //make connection to server socket
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            // read the date from the socket
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            // close the socket connection
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Client read  
Run first





# Remote Procedure Calls

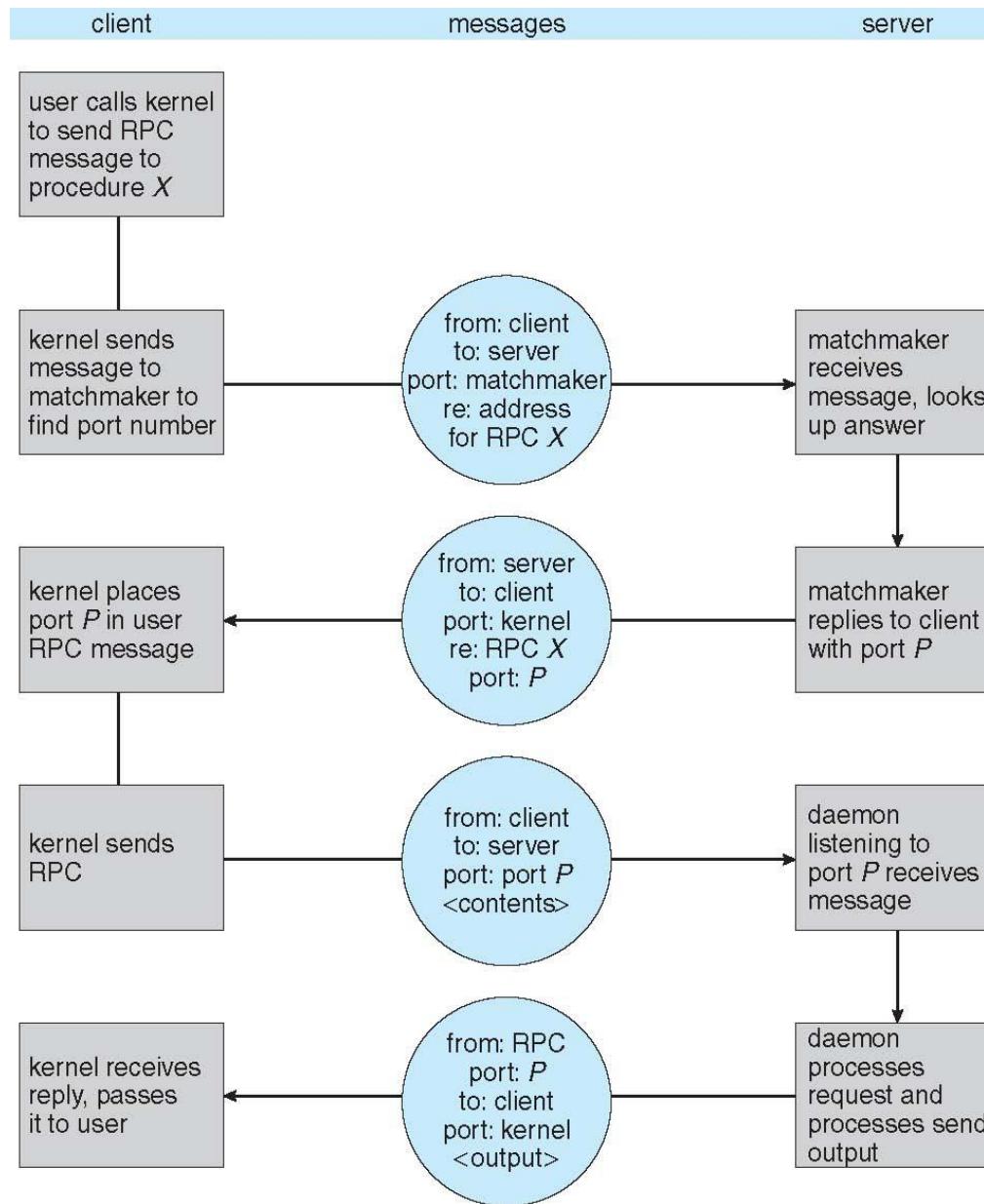
---

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **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





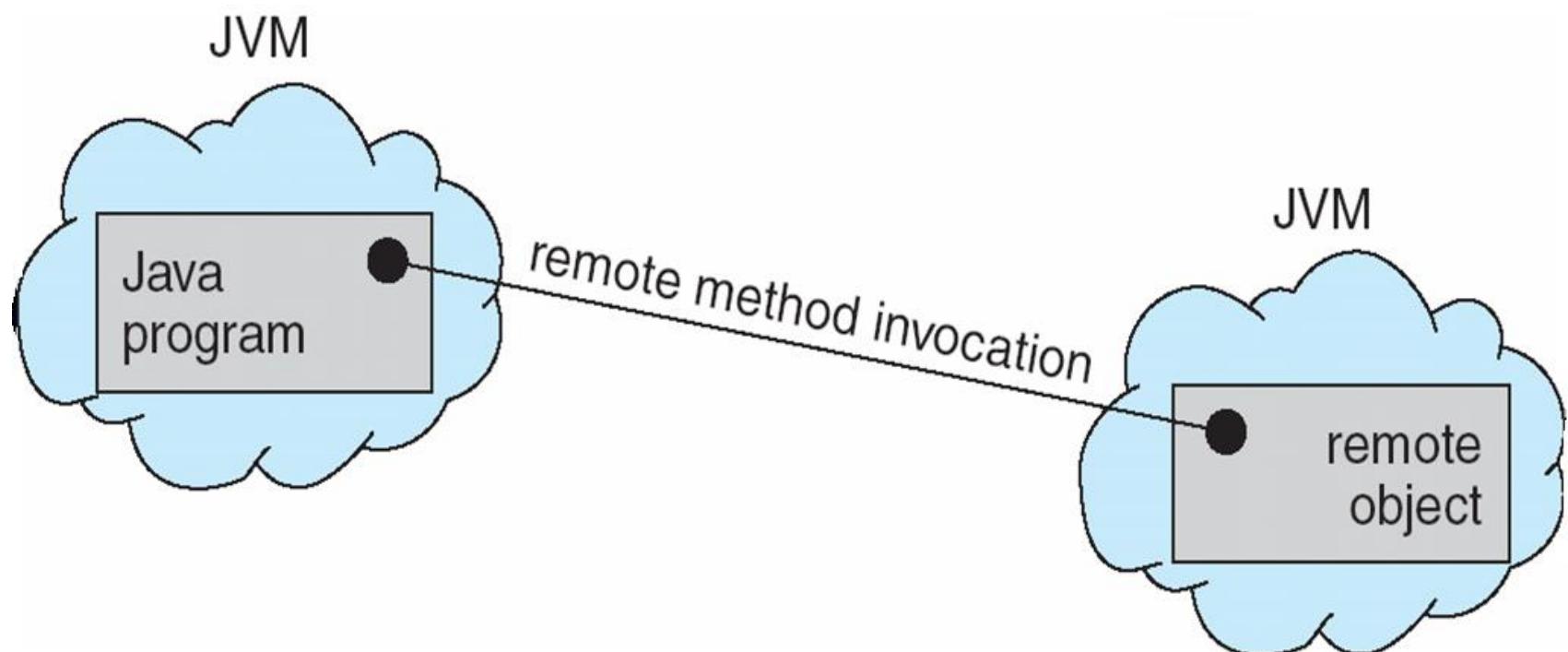
# Execution of RPC





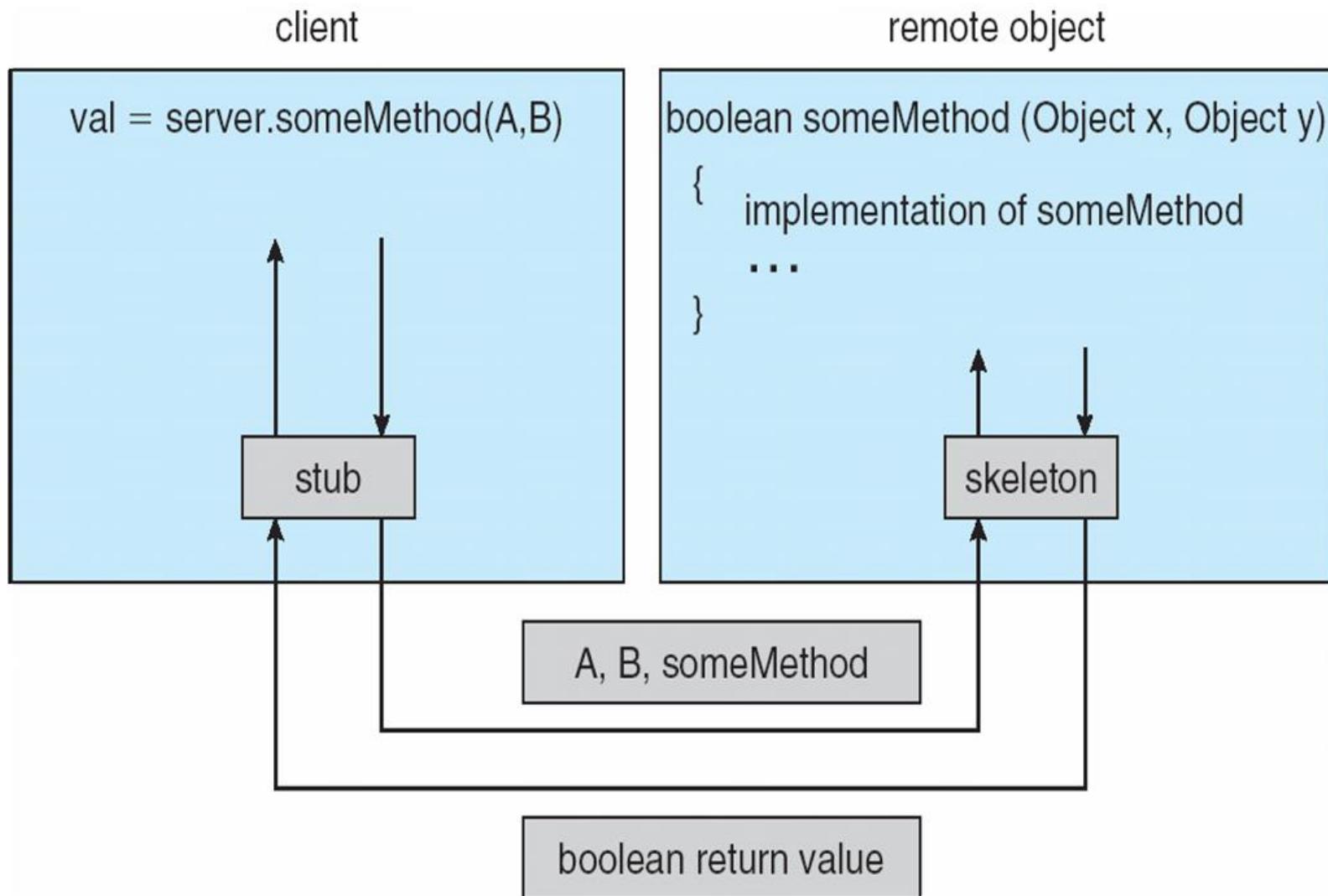
# Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
- RMI allows a Java program on one machine to invoke a method on a remote object





# Marshalling Parameters





# RMI Example

```
public interface RemoteDate extends Remote
{
    public abstract Date getDate() throws RemoteException;
}
```





# RMI Example

```
public class RemoteDateImpl extends UnicastRemoteObject
    implements RemoteDate
{
    public RemoteDateImpl() throws RemoteException { }

    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String[] args) {
        try {
            RemoteDate dateServer = new RemoteDateImpl(); Remote object

                // Bind this object instance to the name "DateServer"
                Naming.rebind("DateServer", dateServer);
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```





# RMI Example

```
public class RMIClient
{
    public static void main(String args[])
    {
        try {
            String host = "rmi://127.0.0.1/DateServer";

            RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
            System.out.println(dateServer.getDate());
        }                                Call remote object
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```



# End of Chapter 3



# Chapter 4: Threads 线程





# Chapter 4: Threads

---

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads





# 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, Win32, and Java thread libraries
- To examine issues related to multithreaded programming





# What's thread

- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.
- Thread is the segment of a process which means a process can have multiple threads and these multiple threads are contained within a process.
- A thread is an execution thread in a program, or A thread is a single sequence stream within a process.
- Threads provide a way for a process to divide (termed "split") itself into two or more simultaneously running tasks. or, multiple threads of execution can be run concurrently by a process.





# Process and thread

- Threads are also called lightweight processes as they possess some of the properties of processes.
- Each thread belongs to exactly one process. The process can create many threads. Thread can not exist independently
- All threads share process resources like code, data, and files can be shared.
- Stacks and registers can't be shared among the threads. Each thread has its own stack and registers.
- Process is basic unit for resource allocation, thread is the basic unit for execution



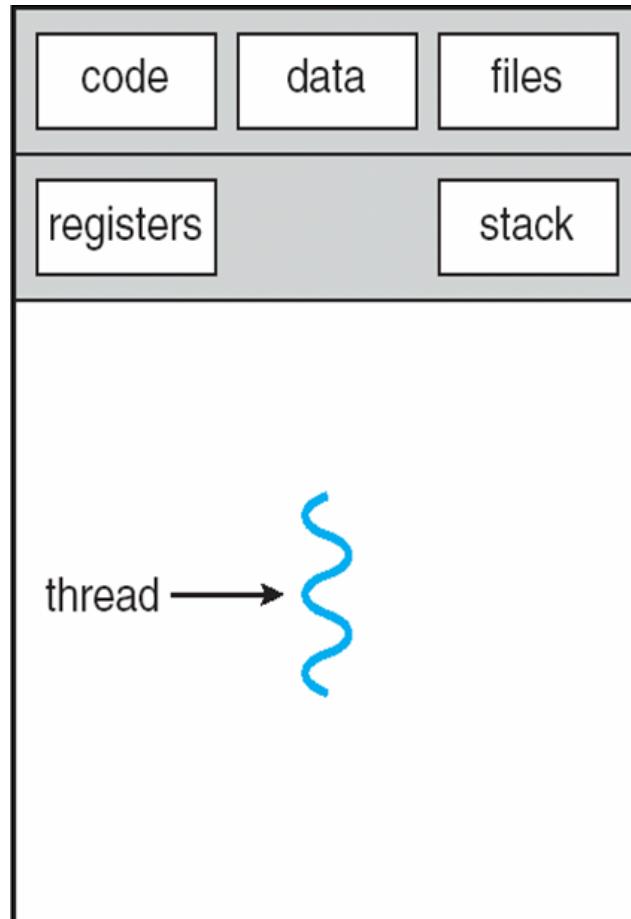


- **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- **Faster context switch:** Context switch time between threads is lower compared to the process context switch. Process context switching requires more overhead from the CPU.
- **Effective utilization of multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
- **Communication:** Communication between multiple threads is easier, as the threads share a common address space. while in the process we have to follow some specific communication techniques for communication between the two processes.
- **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

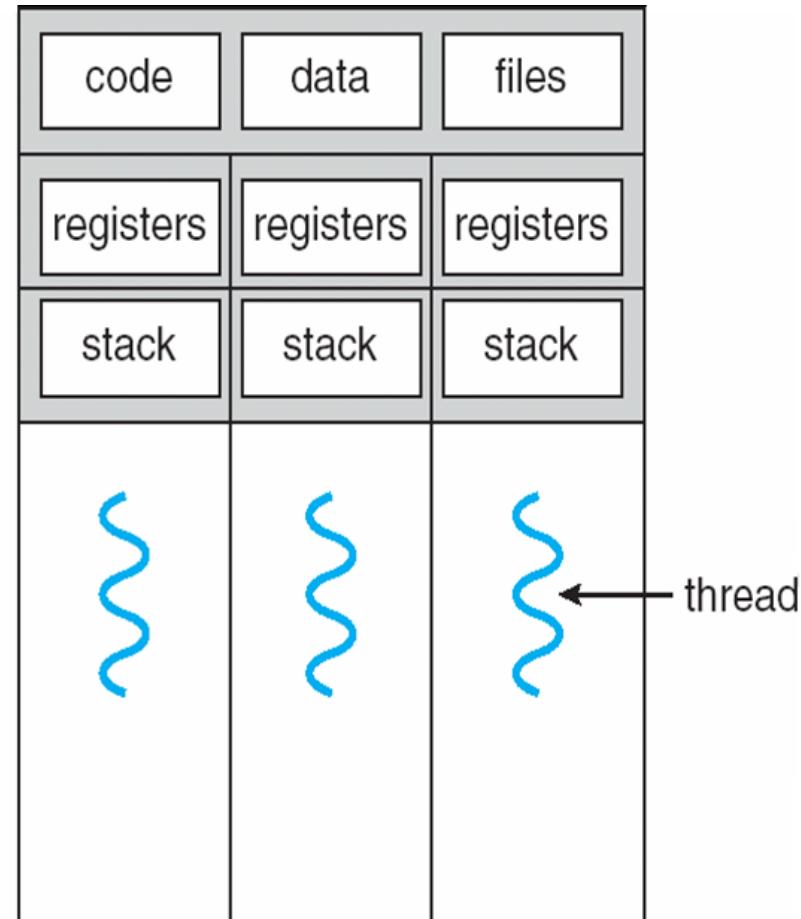




# Single and Multithreaded Processes



single-threaded process

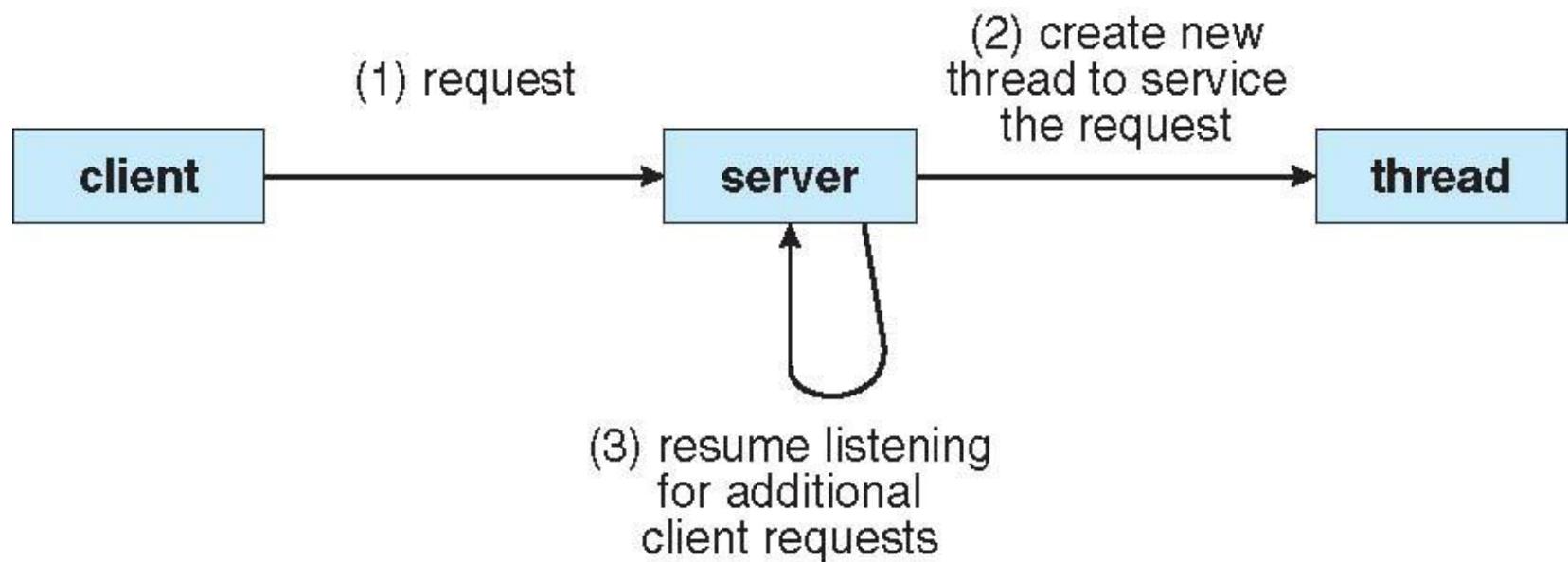


multithreaded process





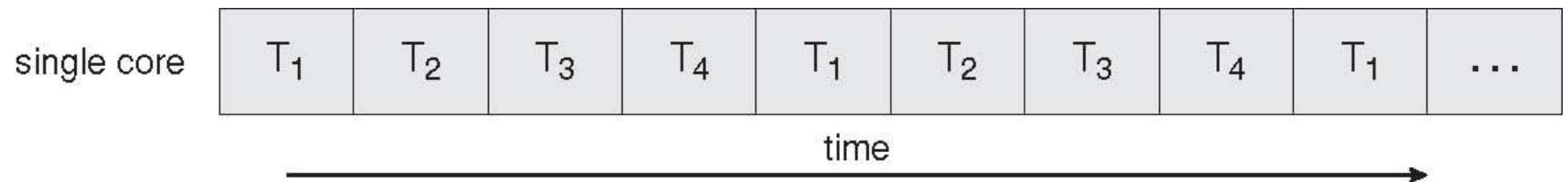
# Multithreaded Server Architecture





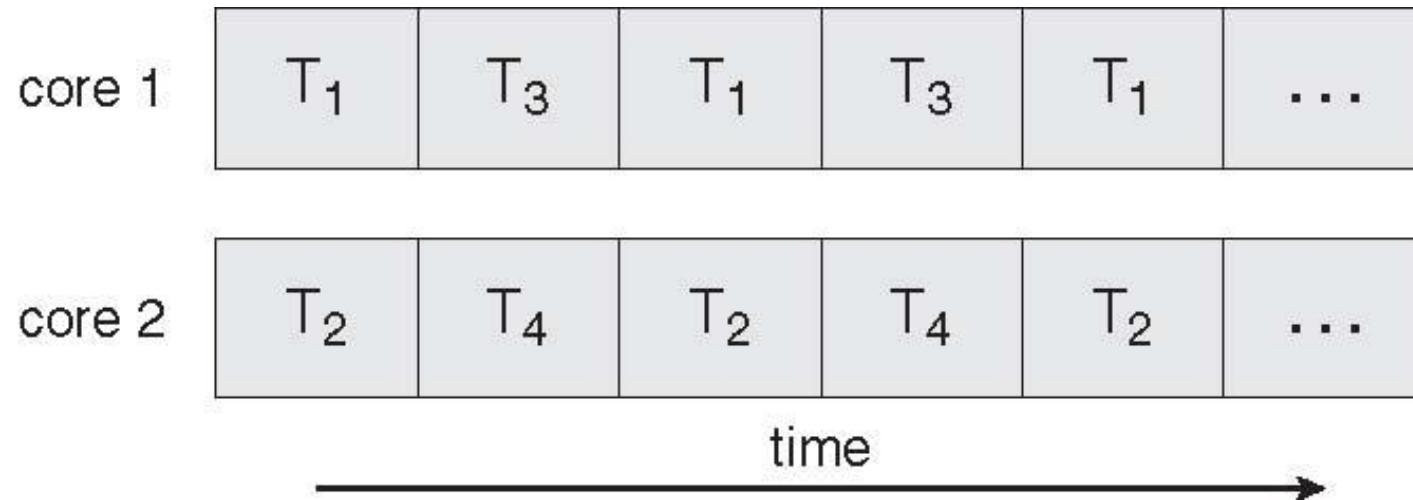
# Concurrent Execution on a Single-core System

并发





# Parallel Execution on a Multicore System





## 并发, Concurrency

■ Two threads are executing in same period

Two threads are overlapping time periods.

Actually, time-slicing

## 并行, Parallel

■ Two threads are executing at same time on a multi-core processor.





# User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads





# Kernel Threads 内核线程

- Supported by the Kernel

- Examples

- Windows XP/2000
- Solaris
- Linux
- Tru64 UNIX
- Mac OS X





# Multithreading Models

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





# Many-to-One

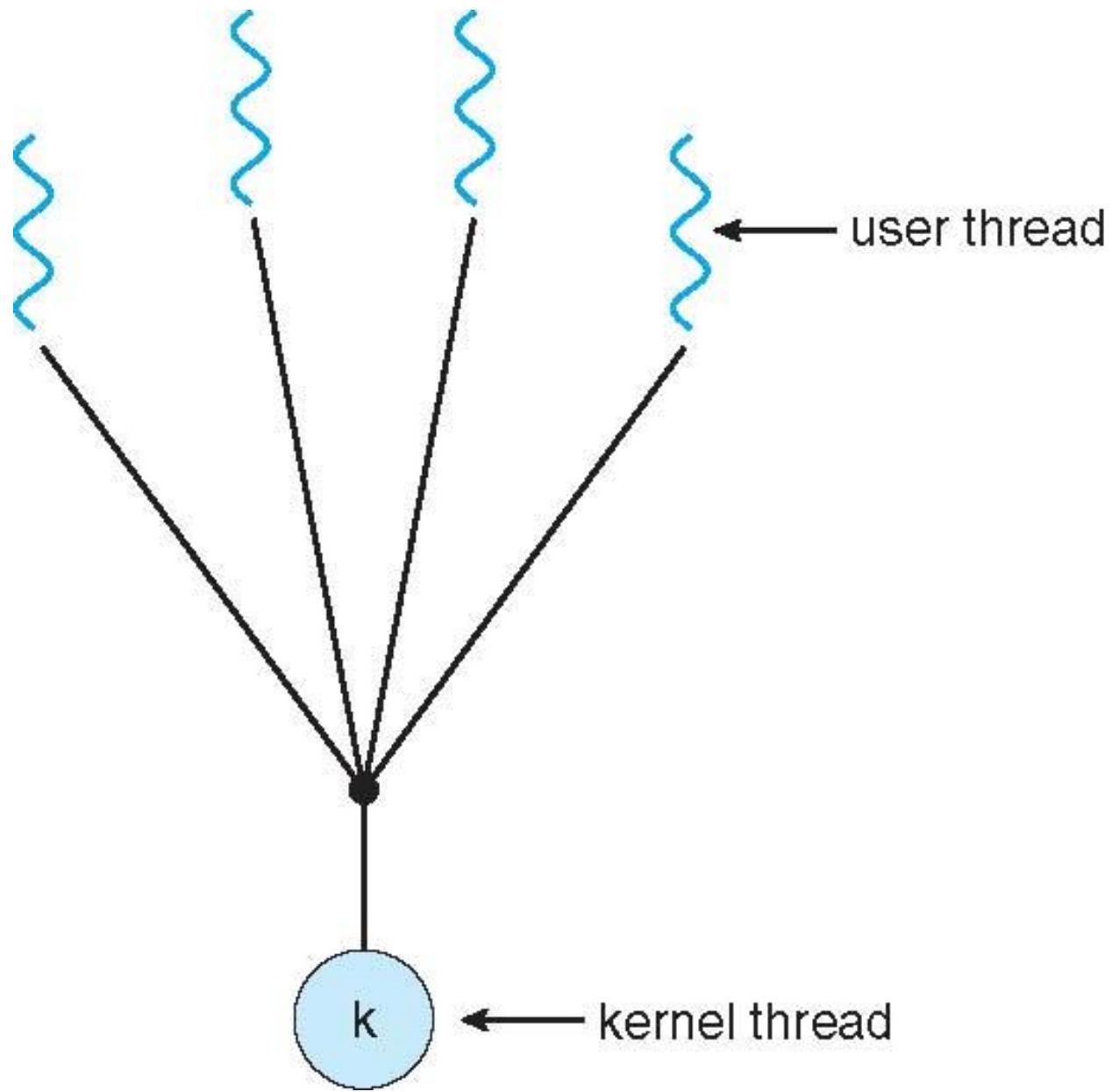
---

- Many user-level threads mapped to single kernel thread
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**





# Many-to-One Model





# One-to-One

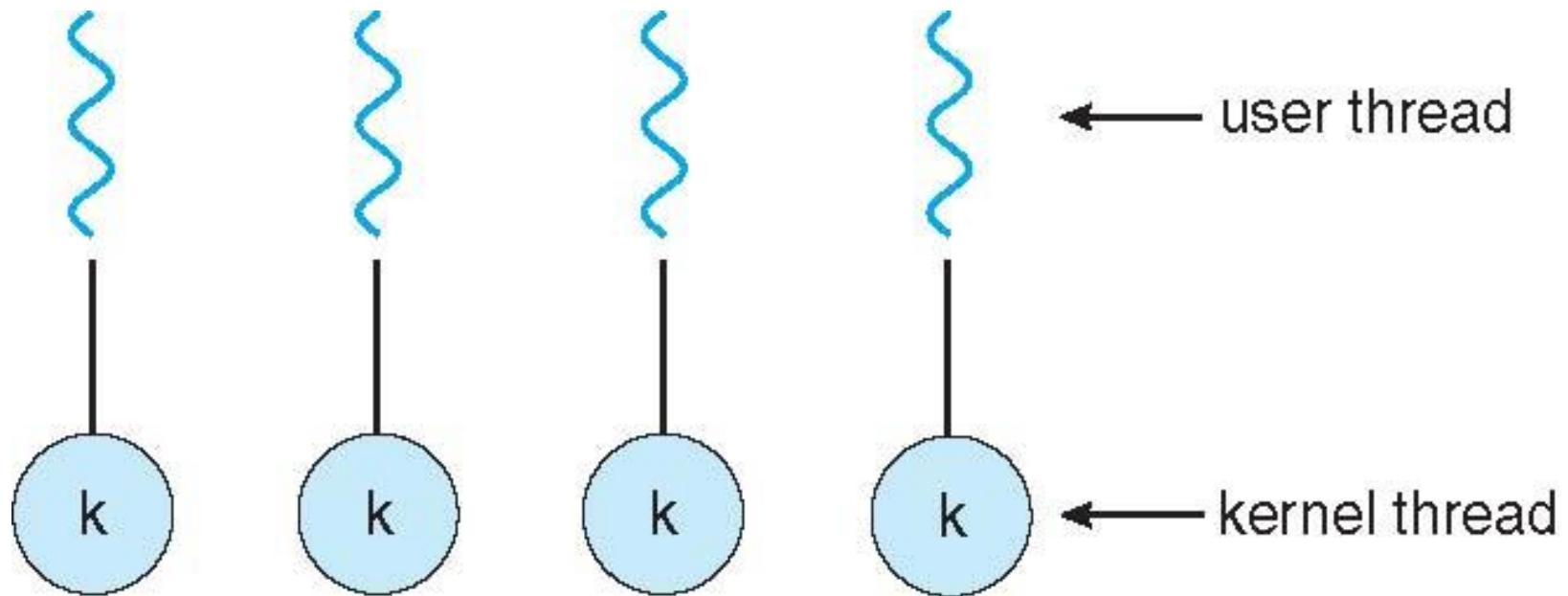
---

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later





# One-to-one Model





# Many-to-Many Model

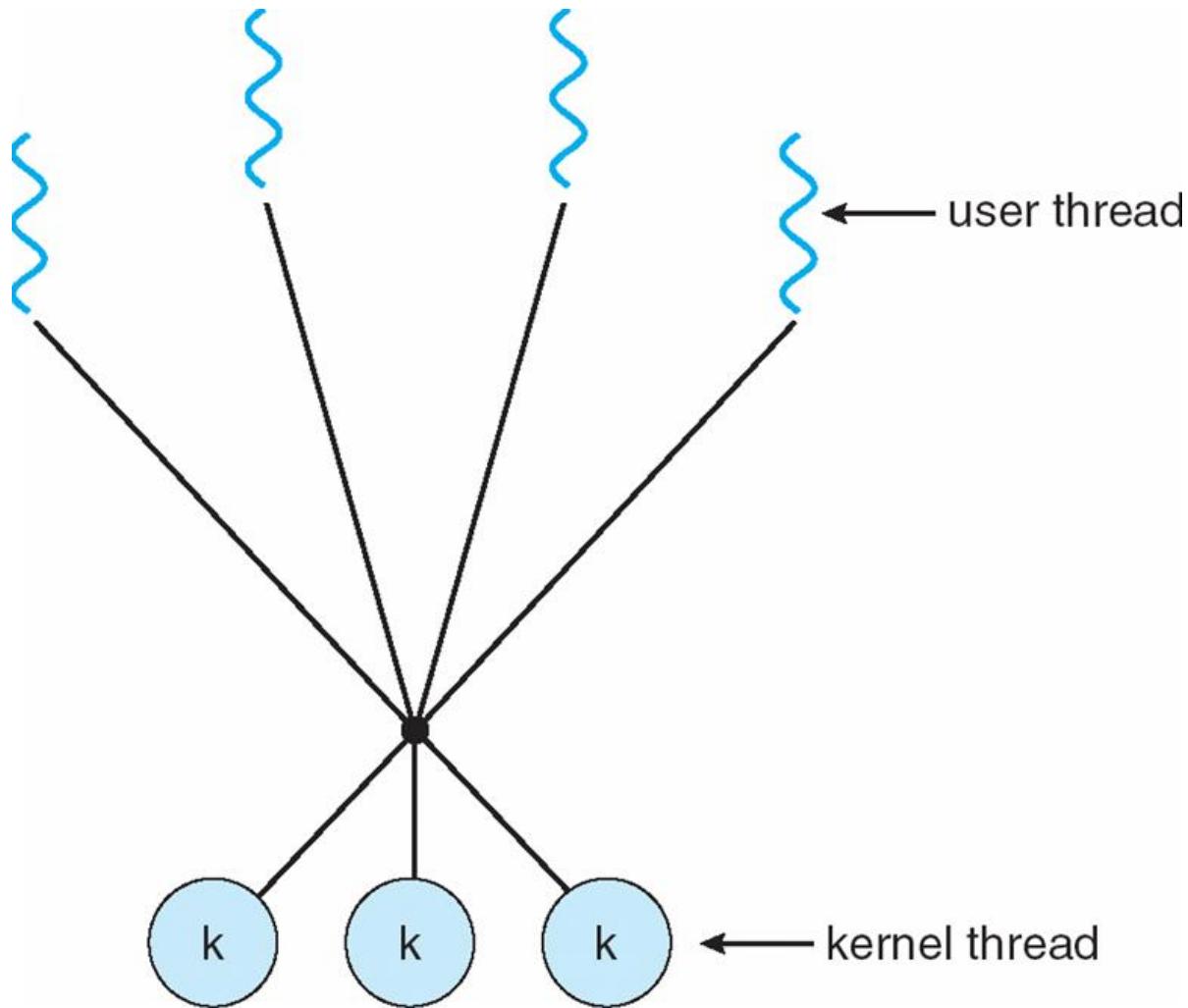
---

- 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 NT/2000 with the *ThreadFiber* package





# Many-to-Many Model





# 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





# Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Implementing the Runnable interface

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





# Java Threads - Example Program

```
class MutableInteger
{
    private int value;
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
}

class Summation implements Runnable
{
    private int upper;
    private MutableInteger sumValue;
    public Summation(int upper, MutableInteger sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }
    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setValue(sum);
    }
}
```

Implement Runnable  
interface

Override: run()  
Thread task





# Java Threads - Example Program

Create thread object

```
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 {
                // create the object to be shared
                MutableInteger sum = new MutableInteger();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sum));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sum.getValue());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>");
    }
}
```

Start thread

Start corresponding thread

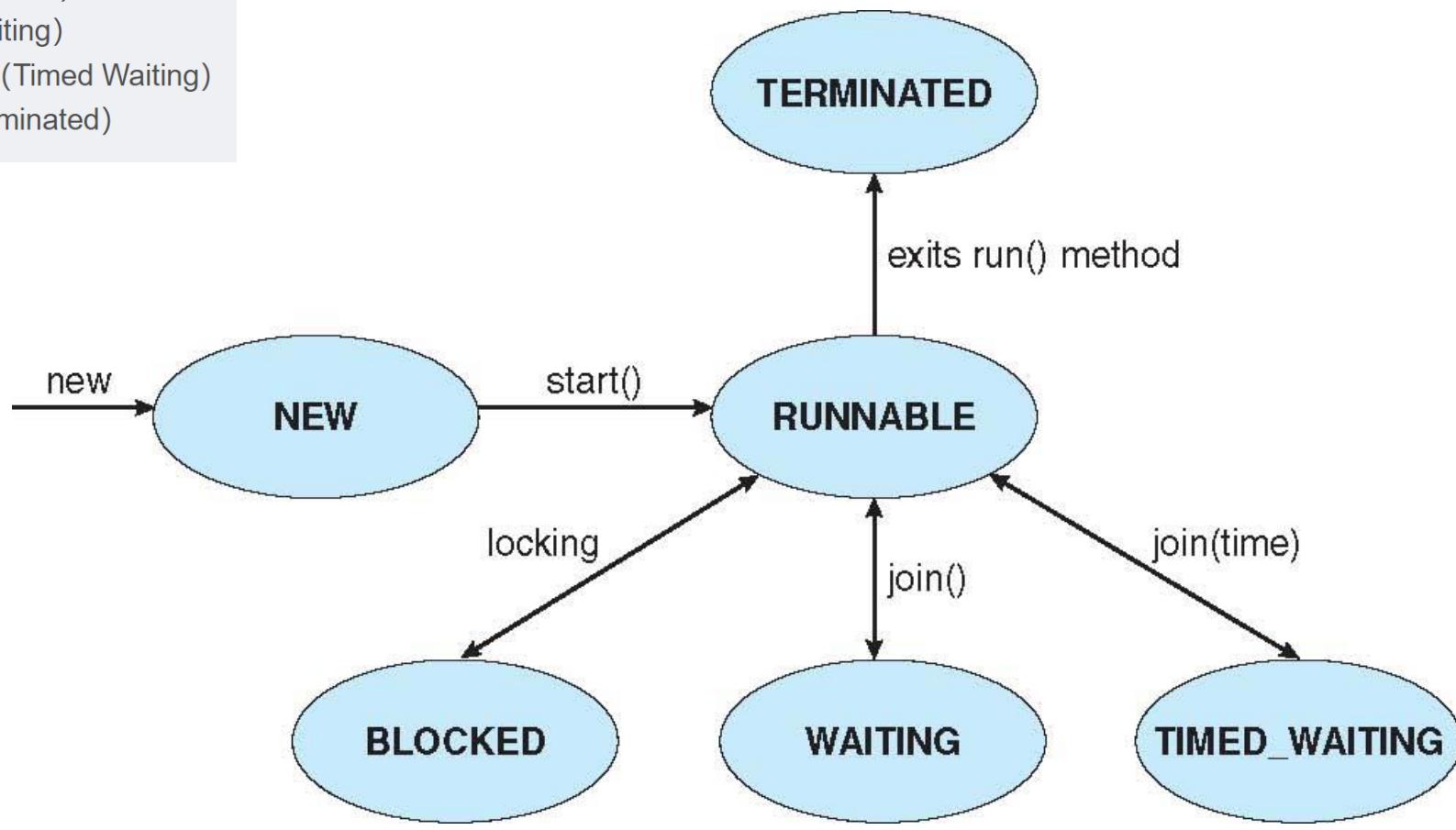
主程序创建线程, 启动线程





# Java Thread States

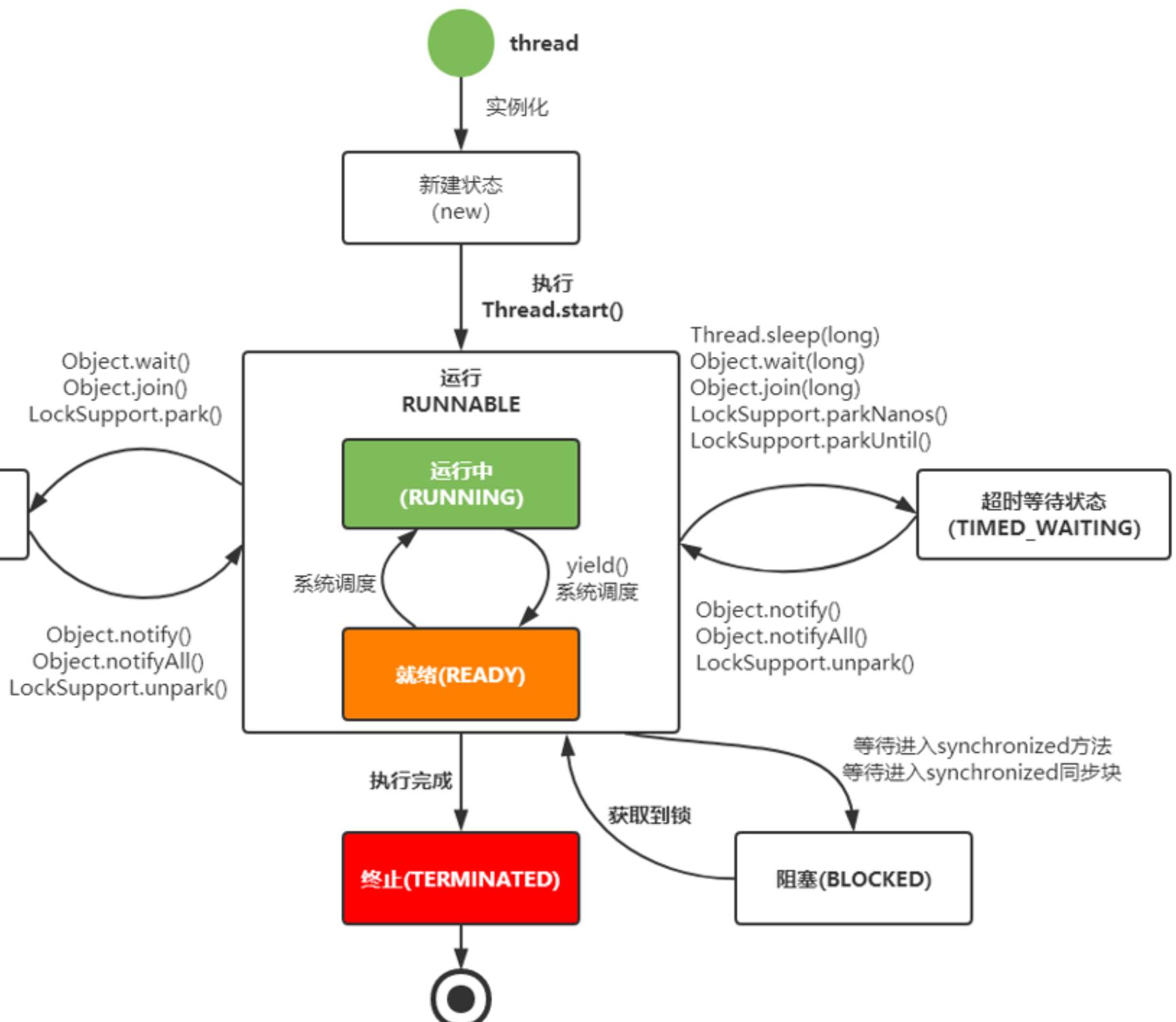
新建状态 (New)  
运行状态 (Runnable)  
阻塞状态 (Blocked)  
等待状态 (Waiting)  
计时等待状态 (Timed Waiting)  
终止状态 (Terminated)



Wait for lock

Wait to be awaken 等待唤醒  
Wait notifying news or interrupt







# Java Threads - Producer-Consumer

```
import java.util.Date;

public class Factory
{
    public static void main(String args[])
    {
        // create the message queue
        Channel<Date> queue = new MessageQueue<Date>();  
          Channel<E>: Pipe  
          传递E类型管道  
          管道对象

        // Create the producer and consumer threads and pass
        // each thread a reference to the MessageQueue object.
        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));

        // start the threads
        producer.start();
        consumer.start();
    }

}
```

Thread synchronization





# Java Threads - Producer-Consumer

```
import java.util.Date;

class Producer implements Runnable
{
    private Channel<Date> queue;      管道消息队列

    public Producer(Channel<Date> queue) {
        this.queue = queue;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();

            // produce an item and enter it into the buffer
            message = new Date();
            System.out.println("Producer produced " + message);
            queue.send(message);
        }
    } } 通过管道发送message
```





# Java Threads - Producer-Consumer

```
import java.util.Date;

class Consumer implements Runnable
{
    private Channel<Date> queue;

    public Consumer(Channel<Date> queue) {
        this.queue = queue;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();

            // consume an item from the buffer
            message = queue.receive();      通过管道接收

            if (message != null)
                System.out.println("Consumer consumed " + message);
        }
    }
}
```





# JAVA Thread class

- public synchronized void start(): 启动一个线程
  - public void run(): 需要重写的方法，方法体就是这个线程要干的事
  - public static native Thread currentThread(): 获取调用该代码的线程
  - public final String getName(): 获取线程的名字
  - public final synchronized void setName(String name): 设置线程的名字
  - public static native void yield(): 释放CPU资源进入就绪状态
  - public final void join(): 在a线程中调用b线程的join()方法会使a线程进入阻塞状态  
直到b线程运行结束
  - public static native void sleep(long millis): 使线程进入阻塞状态millis毫秒
  - public final native boolean isAlive(): 判断一个线程是否存活
  - public final int getPriority(): 获取一个线程的优先级
  - public final void setPriority(int newPriority): 设置线程优先级为newPriority
  - public void wait(): 使线程进入阻塞状态，直到被别的线程调用notify()唤醒
  - public void notify(): 唤醒进入wait状态中优先级最高的线程
  - public void notifyAll(): 唤醒所有进入wait状态的线程





# Threading Issues

- Thread cancellation of target thread

- Thread pools**

- Thread-specific data
- Scheduler activations





# Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled





# 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.





# Thread Specific Data

线程内特定数据

- Allows each thread to have its own copy of data
- Thread-specific data allows a thread to maintain its own global storage that is hidden from the other threads.
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool), i.e. using same variable name for different thread is convenient.





# Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads





# Operating System Examples

---

- Windows XP Threads
- Linux Thread





# Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)



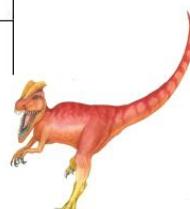
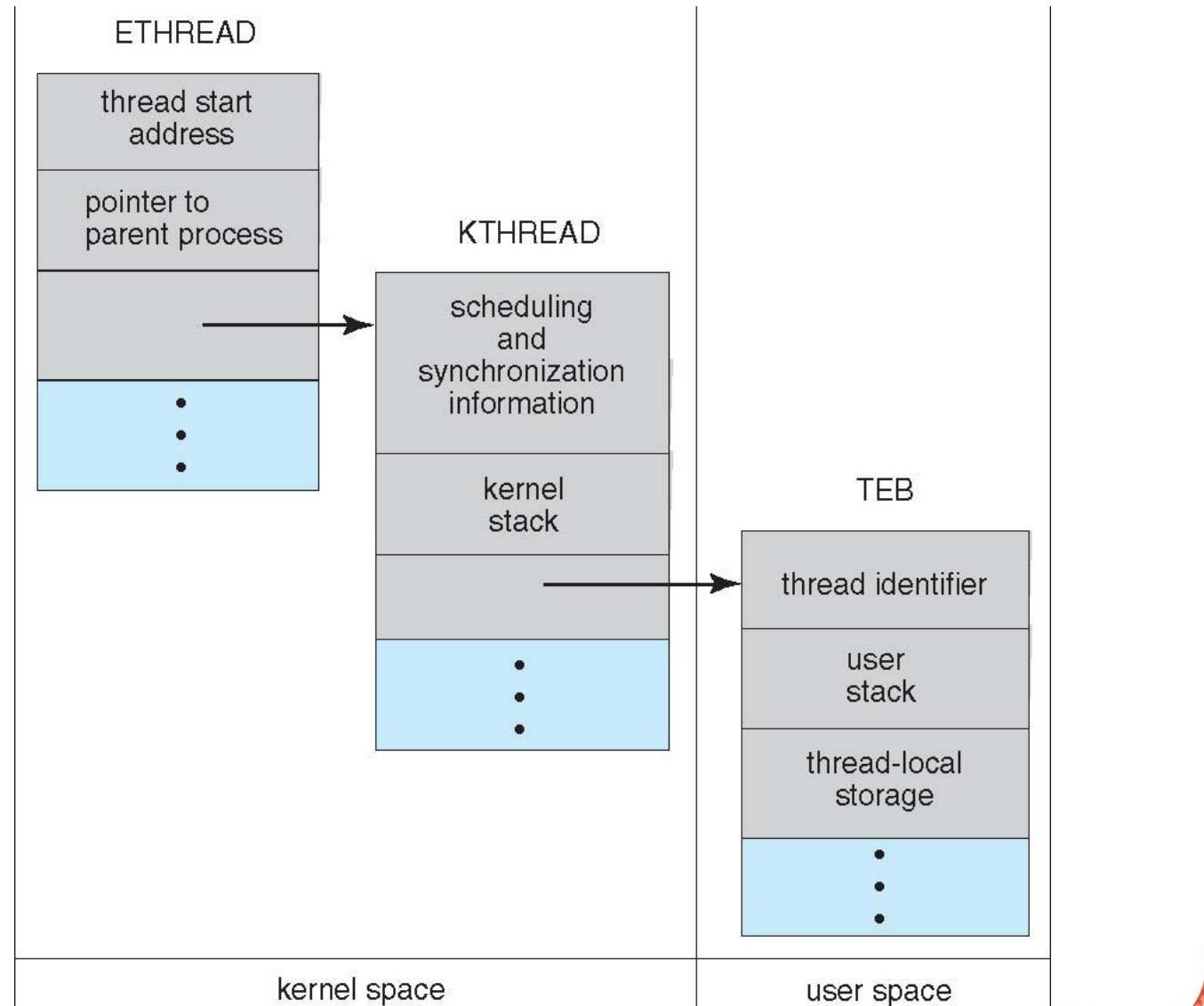


# Windows XP Threads 内核线程

Executive Thread Block  
data structure for execution

Kernel Thread Block  
Thread information in kernel

Thread Environment Block  
Thread in user level





## KTHREAD(内核层线程对象)

```
typedef enum _KTHREAD_STATE {  
    Initialized,  
    Ready,  
    Running,  
    Standby,  
    Terminated,  
    Waiting,  
    Transition  
} KTHREAD_STATE;
```





# Linux Threads

---

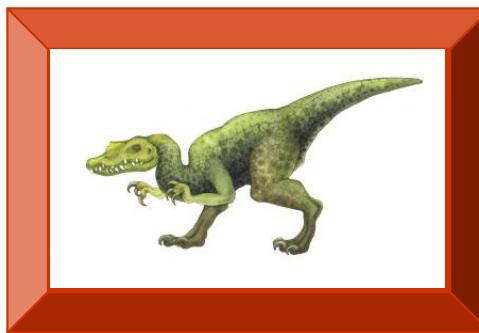
- Linux refers to them **as *tasks*** rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)



# End of Chapter 4



# Chapter 5: CPU Scheduling





# Chapter 5: CPU Scheduling

---

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor 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





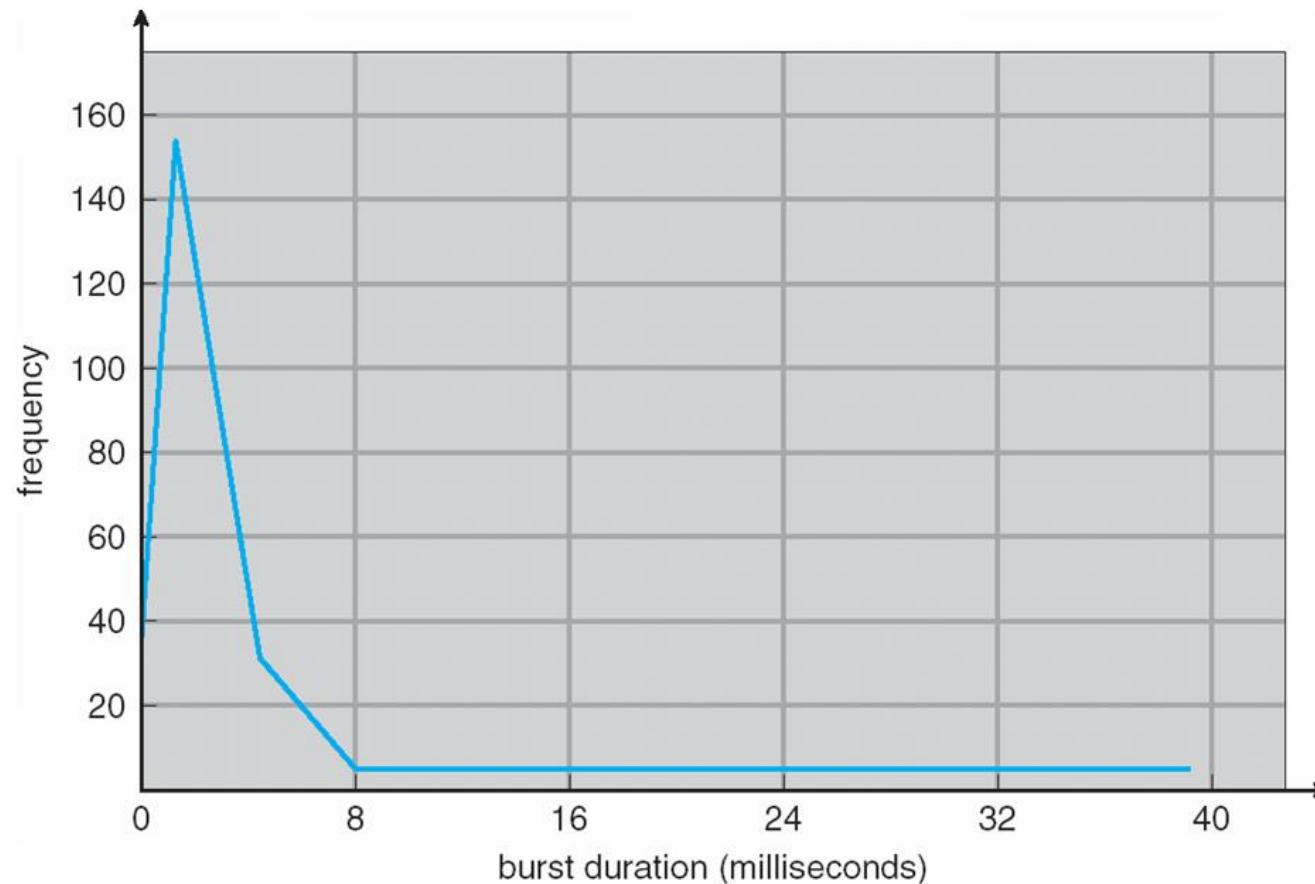
# Basic Concepts

- 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** distribution



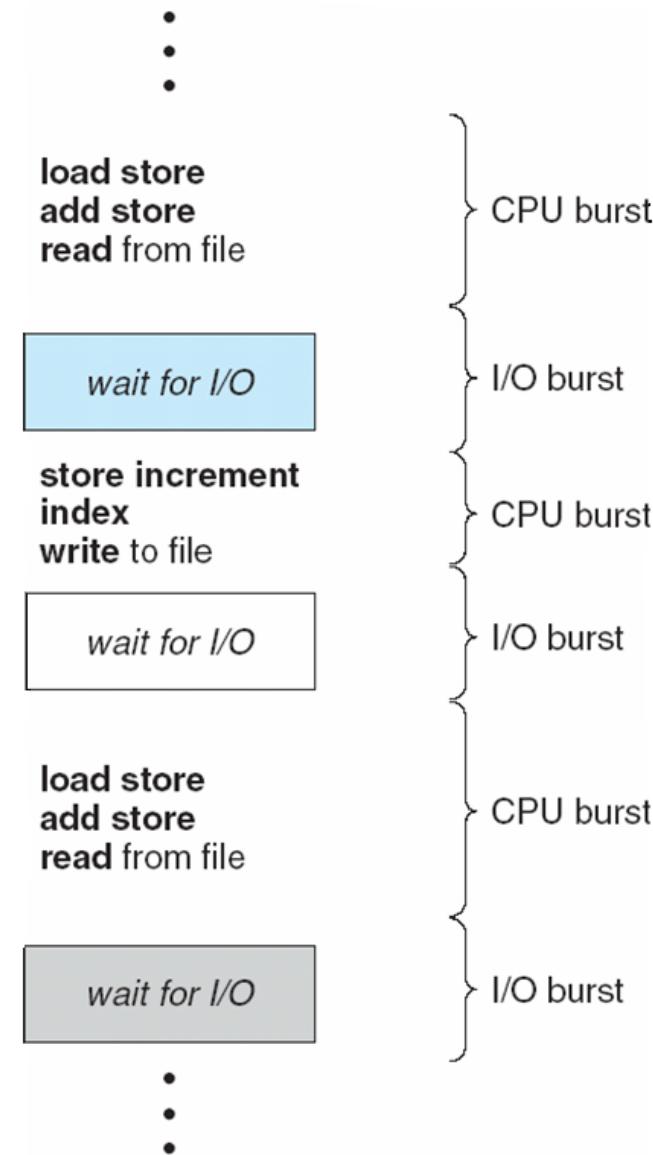


# Histogram of CPU-burst Times





# Alternating Sequence of CPU And I/O Bursts





- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- 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**





- 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

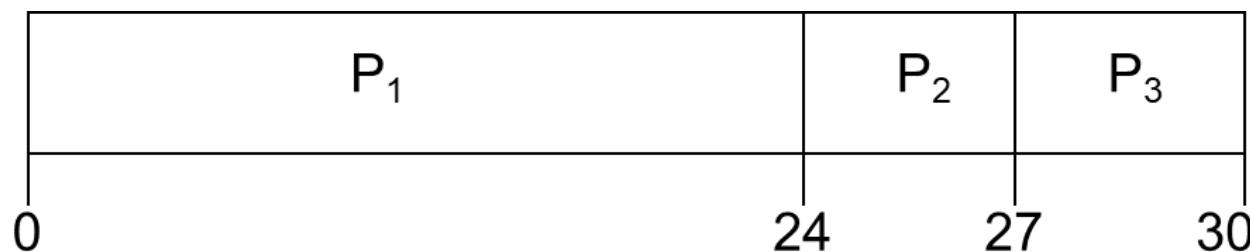
- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

A Gantt chart is a type of bar chart that illustrates a process schedule.





This chart lists the processes to be performed on the vertical axis, and time intervals on the horizontal axis.

The width of the horizontal bars in the graph shows the duration of each process.





# 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





# 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

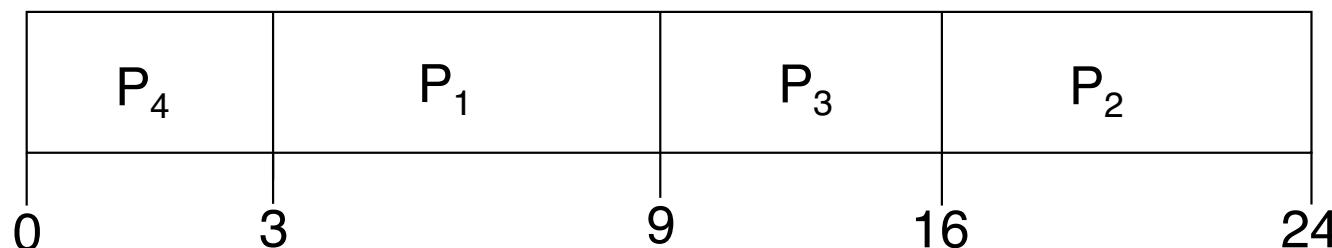




# Example of SJF

<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
- 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 :  $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ .

Note:

Assume CPU takes more rounds

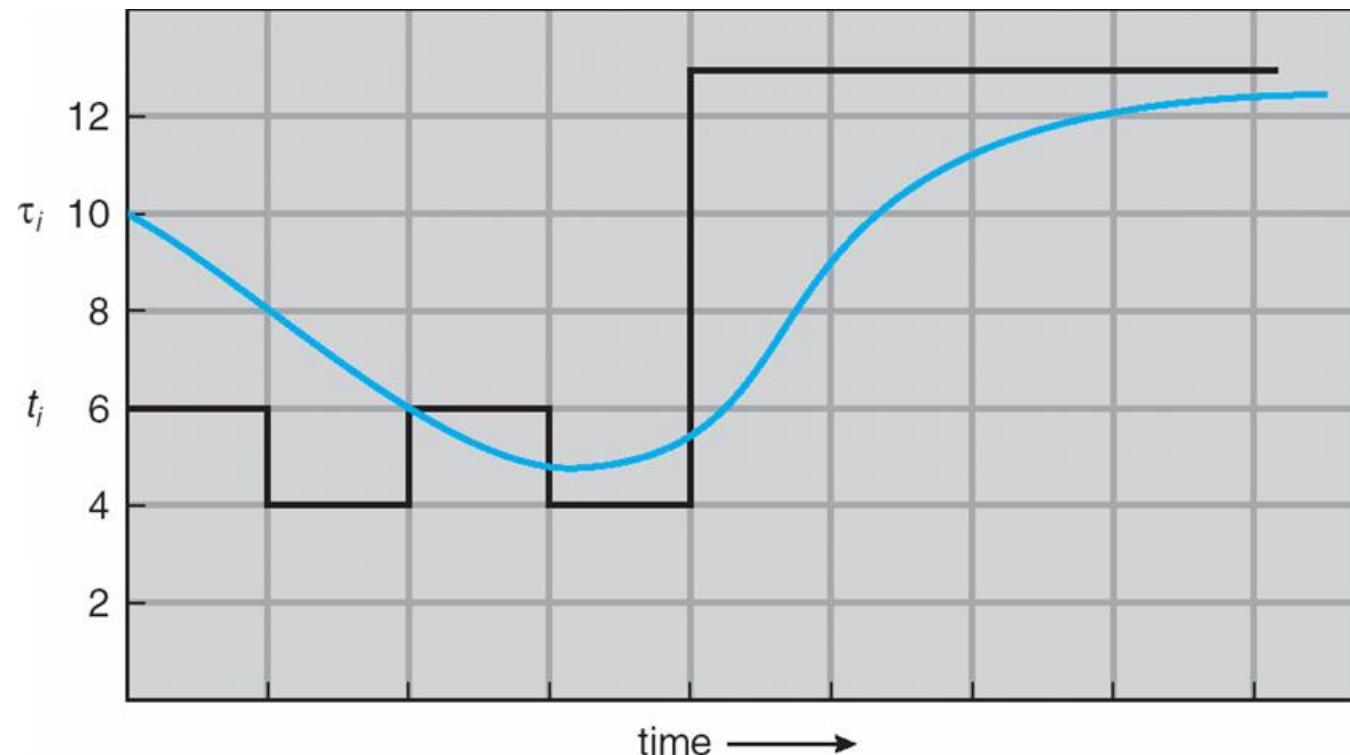
Initial or previous round t is known

So the each process in later round can be calculated





# 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

schedule the process with the shortest time  
4 ,6, 13 are selected in next round  
p2,p4,p1,p3,p6,p7,p8





# 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:

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

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





# 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 a priority scheduling where priority is the 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





# Round Robin (RR)

- Round Robin(Polling, 轮询调度)
- Each process gets a small unit of CPU time (*time quantum*), 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.
- Performance with  $q$ 
  - $q$  large  $\Rightarrow$  FIFO or FCFS
  - $q$  small  $\Rightarrow$   $q$  must be large with respect to context switch, otherwise overhead is too high

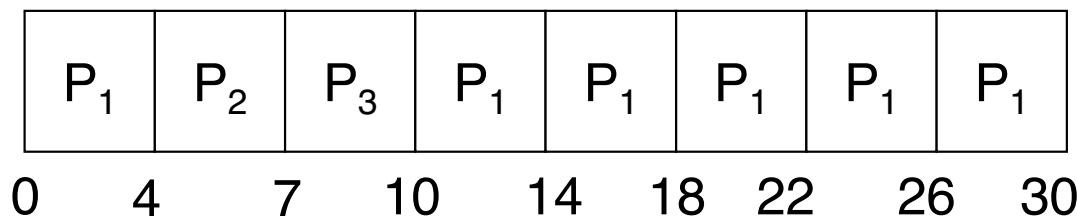




# Example of RR with Time Quantum = 4

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

- The Gantt chart is:

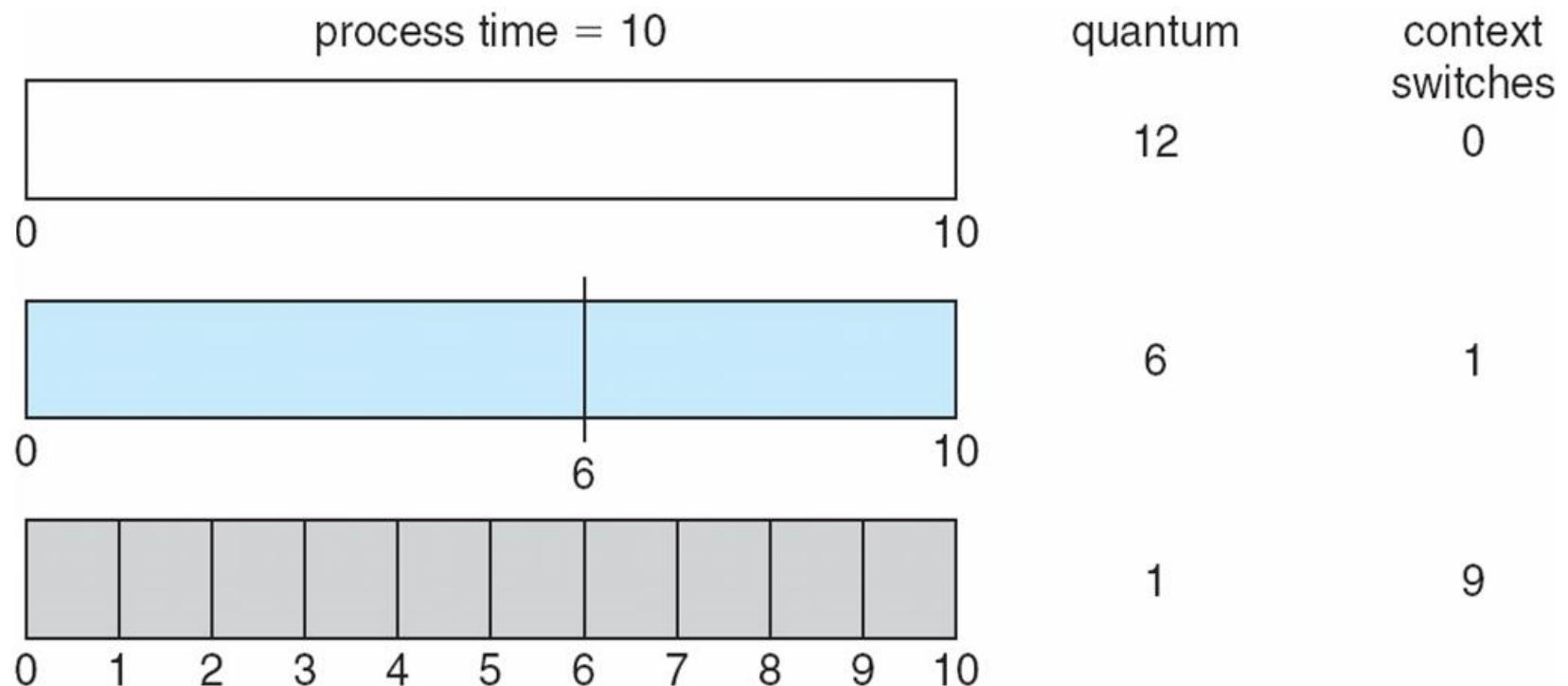


- Typically, higher average turnaround than SJF, but better *response*



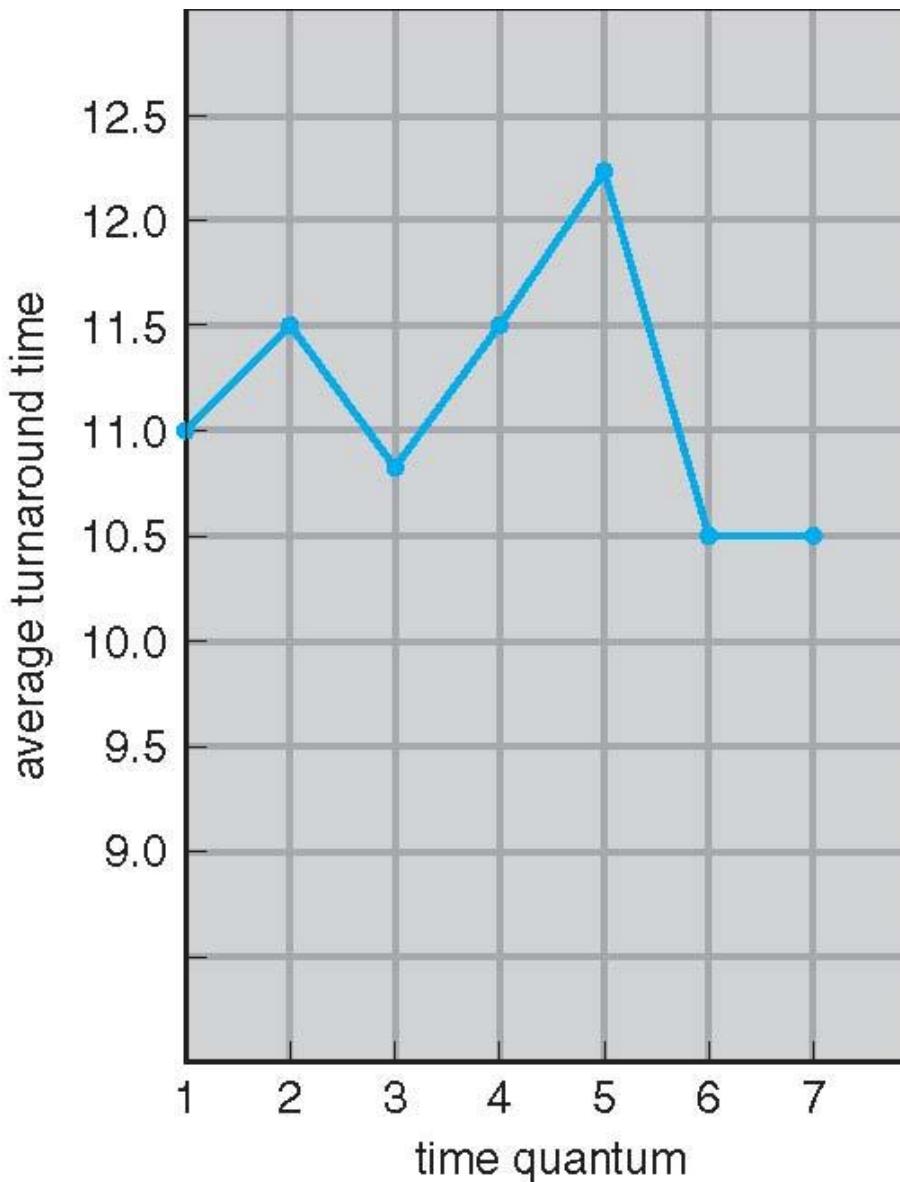


# Time Quantum and Context Switch Time





# Turnaround Time Varies With The Time Quantum



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





# Multilevel Queue

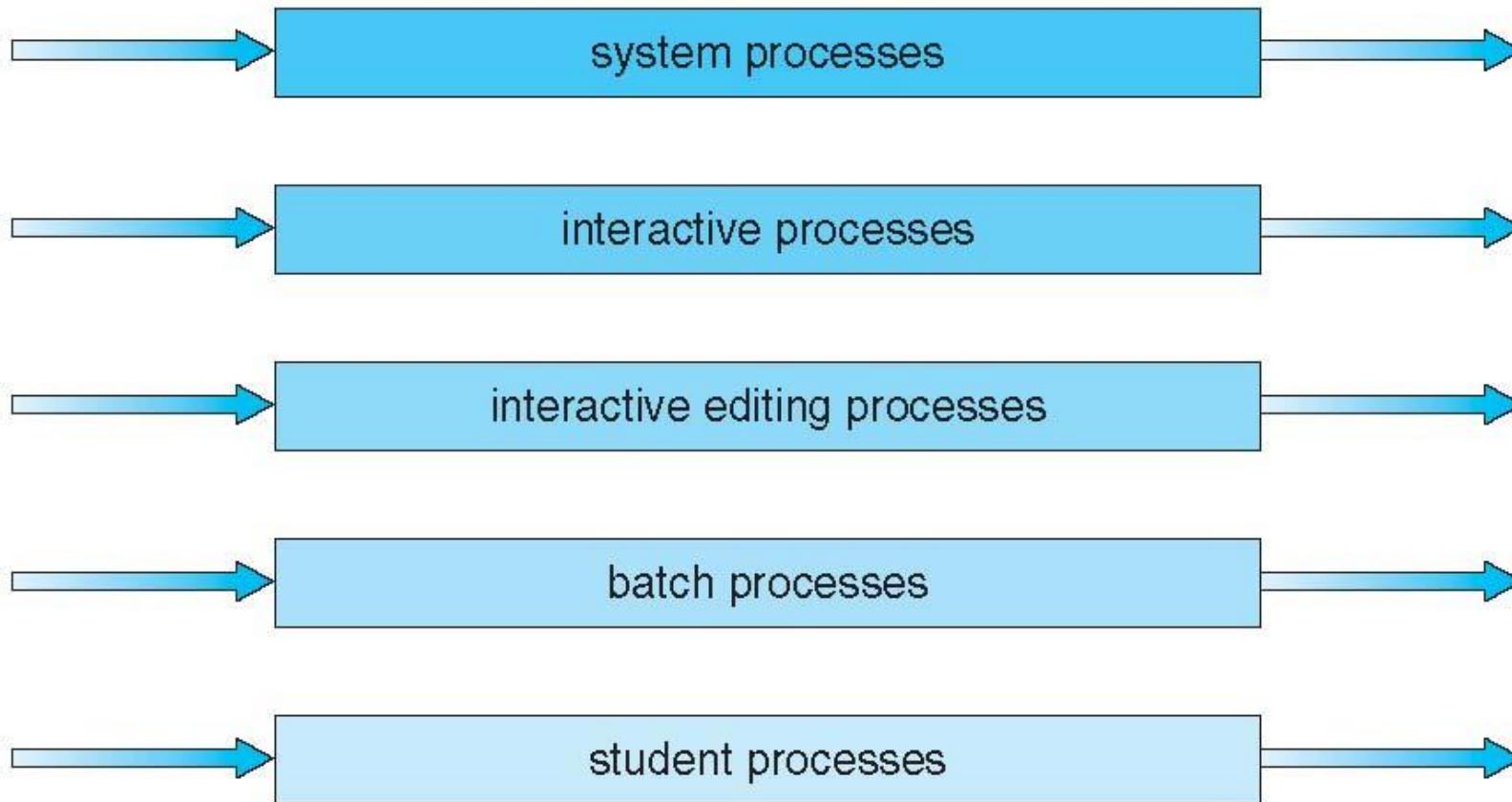
- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then 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

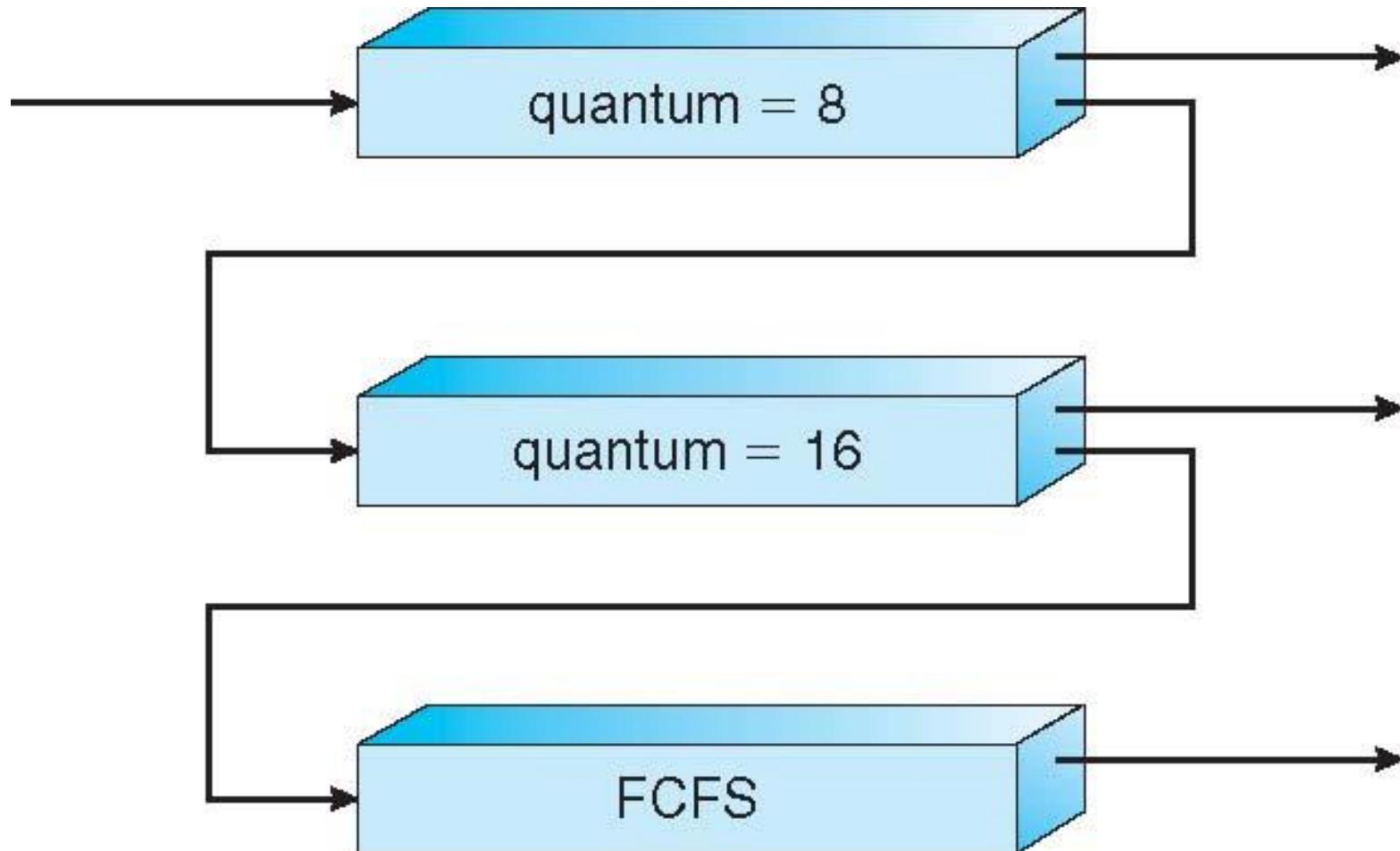
- Scheduling

- 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  $Q_2$ .





# Multilevel Feedback Queues





# Thread Scheduling

- Distinction between user-level and kernel-level threads
- 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
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system





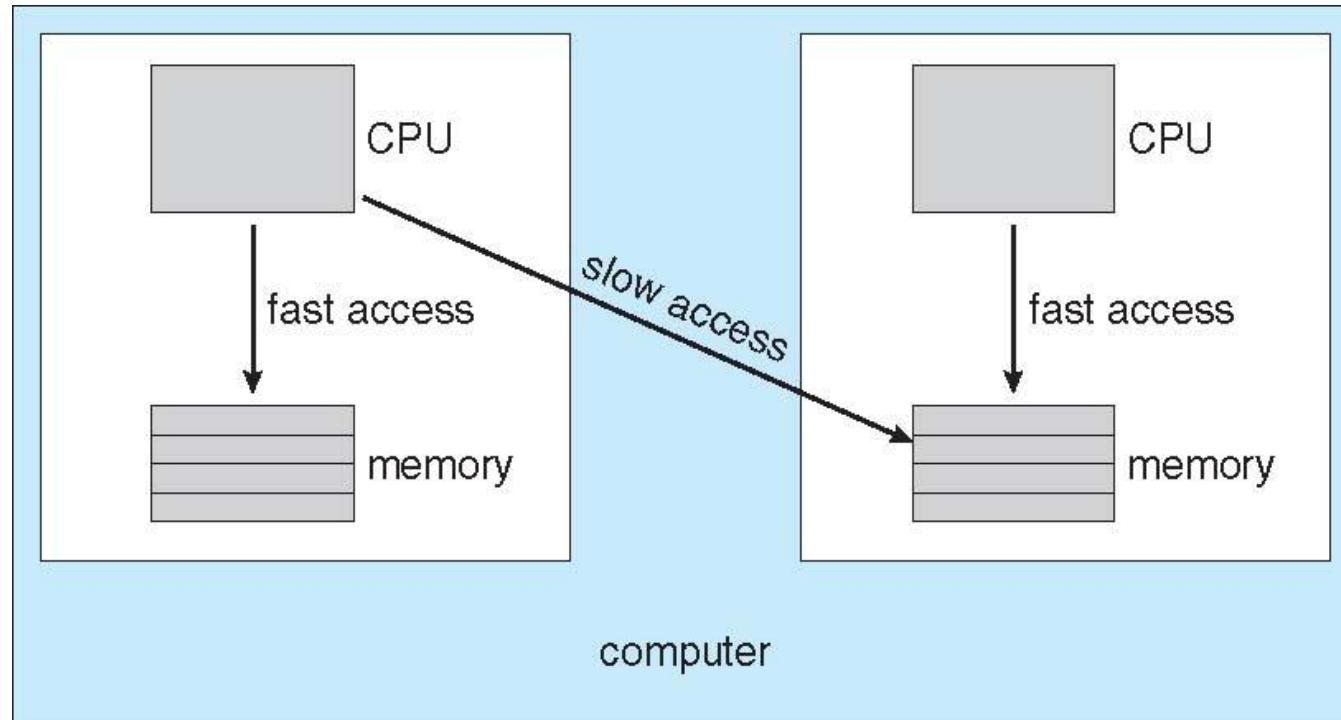
# Multiple-Processor Scheduling

- 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
  - Load balanced,
  - Push migration and pull migration
- **Processor affinity(亲和性)** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**





# NUMA and CPU Scheduling



non-uniform memory access (NUMA)

Memory frame can affect affinity





# Multicore Processors

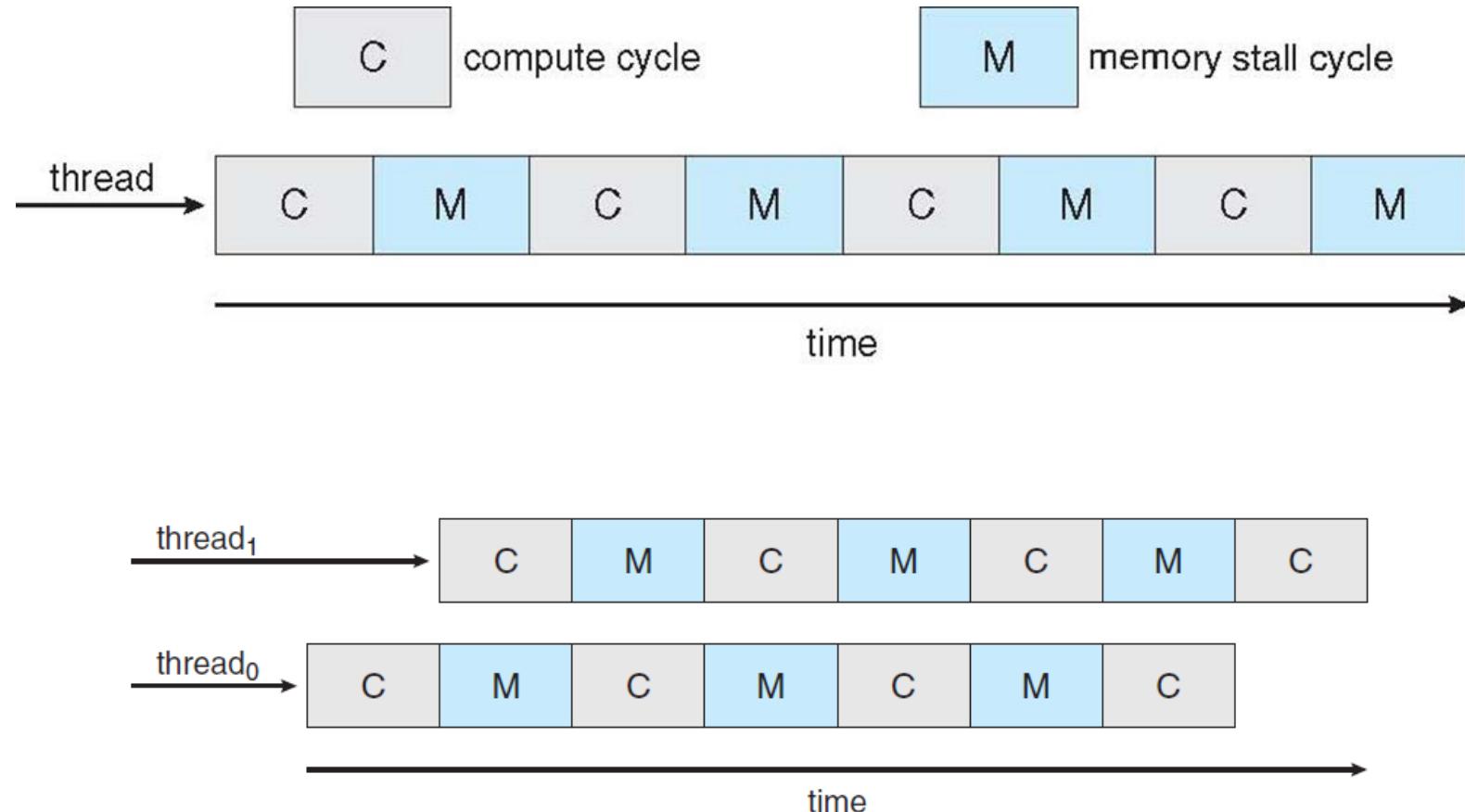
---

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
  - Memory stall means memory cache miss, CPU pause





# Multithreaded Multicore System



**Figure 5.13** Multithreaded multicore system.





# Operating System Examples

---

- Windows XP scheduling
- Linux scheduling





# Windows XP 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





# Linux Scheduling

---

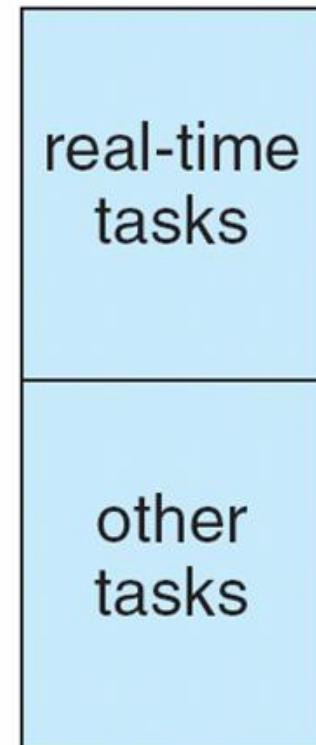
- Constant order  $O(1)$  scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- (figure 5.15)





# Priorities and Time-slice length

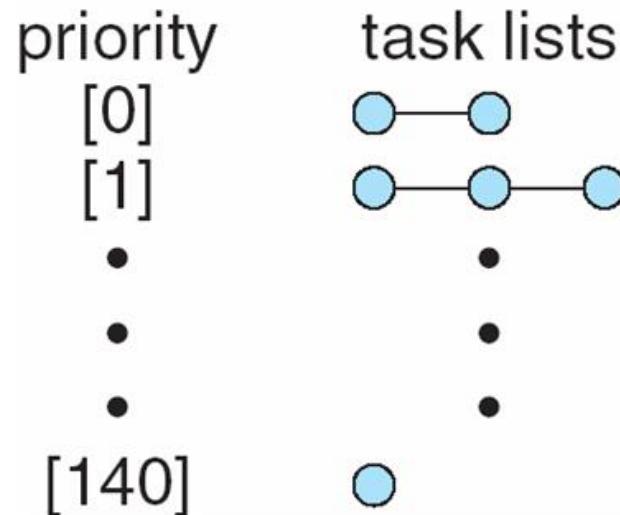
numeric priority	relative priority	time quantum
0	highest	200 ms
•		
•		
•		
99		
100		
•		
•		
•		
140	lowest	10 ms



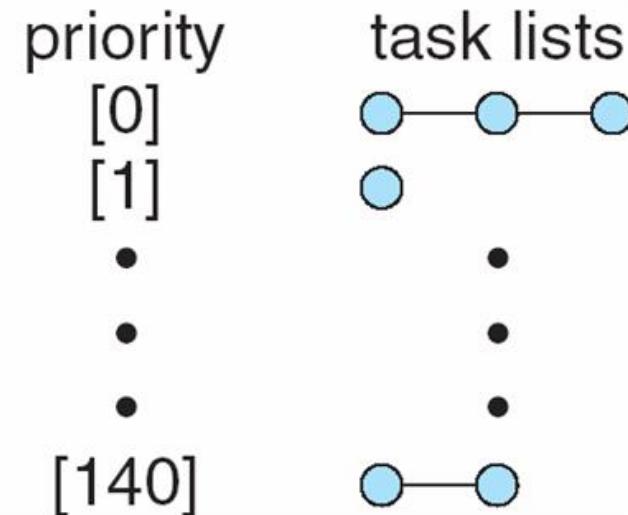


# List of Tasks Indexed According to Priorities

**active  
array**



**expired  
array**





# Algorithm Evaluation

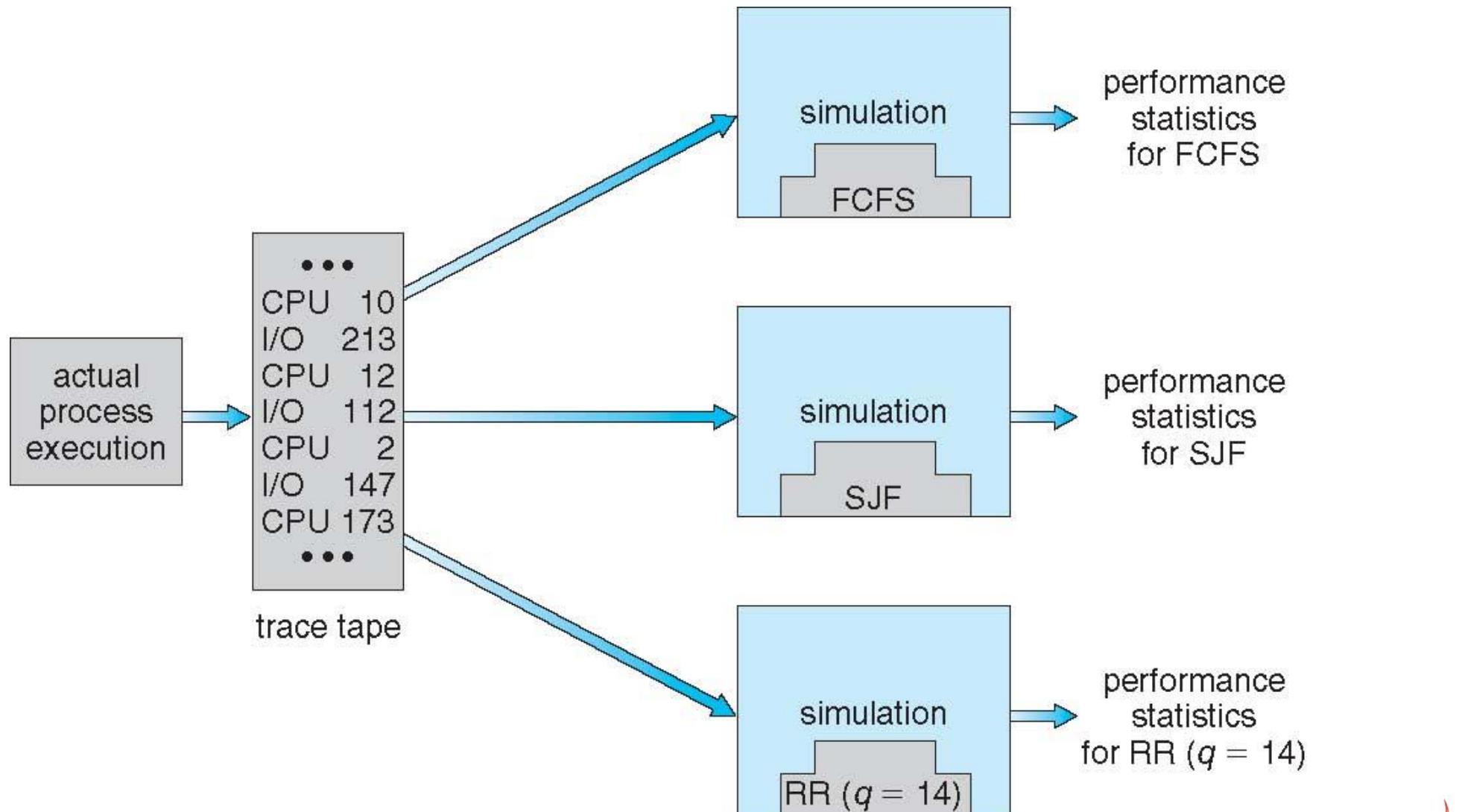
---

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload analytical method
- Queueing models: use model to estimate some performance
- Implementation: Simulation





# Evaluation of CPU schedulers by Simulation





# Java Thread Scheduling

---

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority





# Java Thread Scheduling (Cont.)

---

- JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

\* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not





# Time-Slicing

- Since the JVM Doesn't Ensure Time-Slicing, the `yield()` Method May Be Used:

```
while (true) {  
    // perform CPU-intensive task  
    . . .  
    Thread.yield();  
}
```

- This Yields Control to Another Thread of Equal Priority





# Thread Priorities

<u>Priority</u>	<u>Comment</u>
Thread.MIN_PRIORITY	Minimum Thread Priority
Thread.MAX_PRIORITY	Maximum Thread Priority
Thread.NORM_PRIORITY	Default Thread Priority

Priorities May Be Set Using `setPriority()` method:

```
setPriority(Thread.NORM_PRIORITY + 2);
```

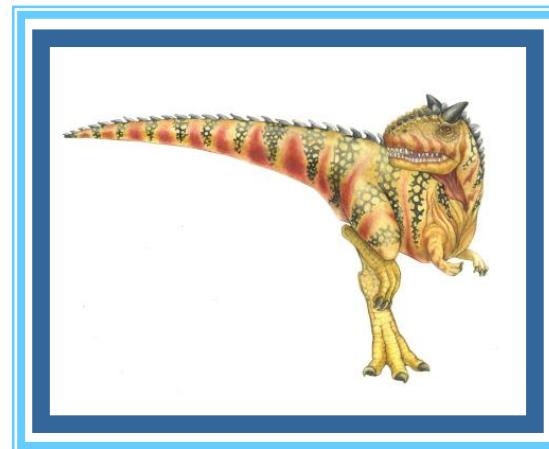


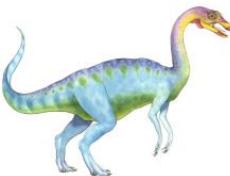
# End of Chapter 5



Exercise

# Chapter 6: Synchronization Tools



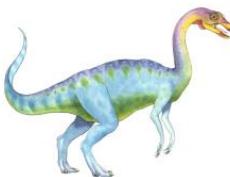


# Outline

---

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Java Synchronization



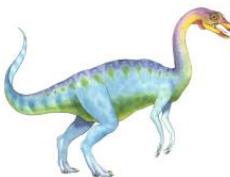


# Objectives

---

- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem





# Background

---

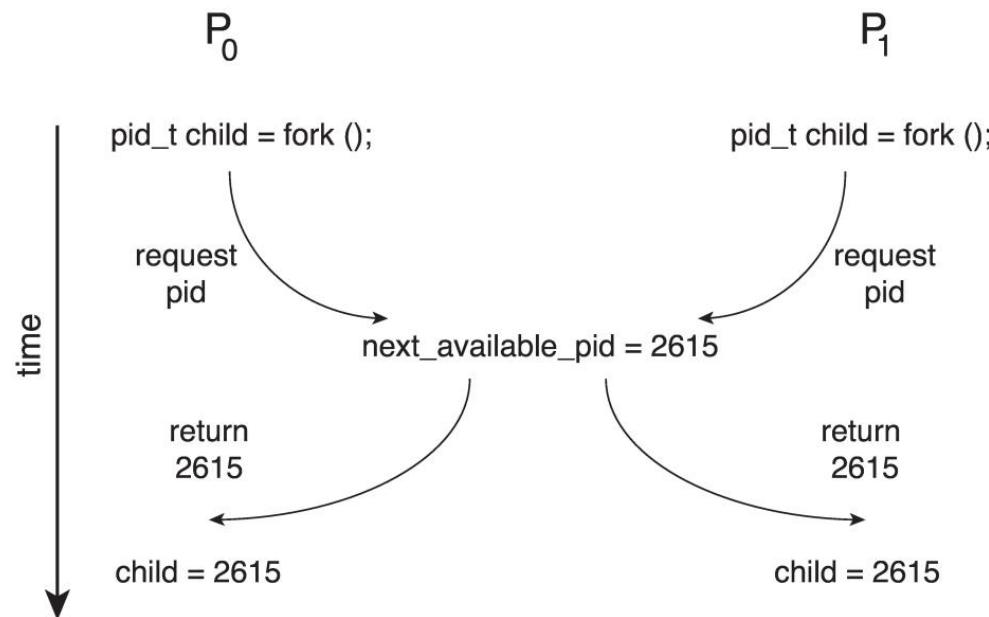
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated in chapter 4 the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.





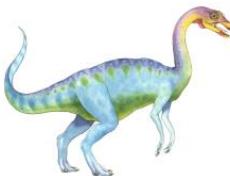
# Race Condition

- Processes  $P_0$  and  $P_1$  are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent  $P_0$  and  $P_1$  from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

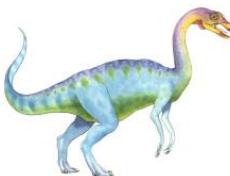




- Thread1 and Thread2 all do i++

- int i=5;
- public void run(){
  - i++;
  - }
  - After one round running
  - i = ?



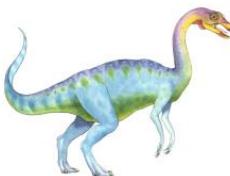


# Critical Section Problem

---

- Consider system of  $n$  processes  $\{p_0, p_1, \dots p_{n-1}\}$
- Each process has **critical section**(临界区) segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**
- **entry section**进入区(入口)
- **exit section**, 退出区(出口)
- **Remainder section**, 剩余区





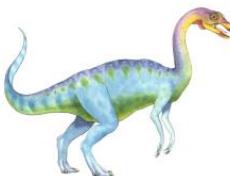
# Critical Section

---

- General structure of process  $P_i$ ,

```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```





# Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

**1. Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections

## **2. Progress**

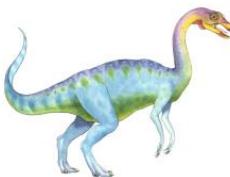
- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely

### **1. Bounded Waiting(有限等待)**

- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the  $n$  processes



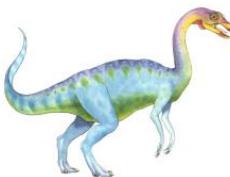


# Interrupt-based Solution

---

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?



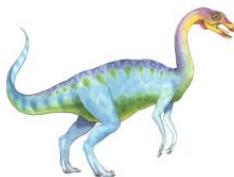


# Software Solution 1

---

- Two process solution: Pi and Pj
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **int turn;** Shared variable
- The variable **turn** indicates whose turn it is to enter the critical section
- initially, the value of **turn** is set to *i*





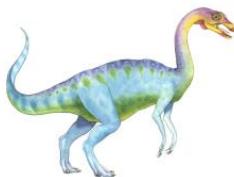
# Algorithm for Process $P_i$

```
while (true) {  
  
    while (turn == j);  
  
    /* critical section */  
  
    turn = j;  
  
    /* remainder section */  
  
}
```

$P_i$  wait and let  $P_j$  run

Let  $P_j$  run



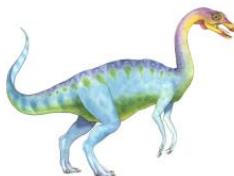


# Algorithm for Process $P_j$

```
while (true) {  
    while (turn == i);  
    /* critical section */  
    turn = i;  
    /* remainder section */  
}
```

$P_j$  wait and let  $P_i$  run





# Correctness of the Software Solution

---

- Mutual exclusion is preserved

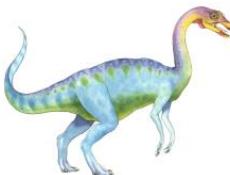
$P_i$  enters critical section only if:

**turn = i**

and **turn** cannot be both 0 and 1 at the same time

- What about the Progress requirement?
- What about the Bounded-waiting requirement?
- What if  $P_i$  or  $P_j$  are not ready for running, e.g. additional condition is required



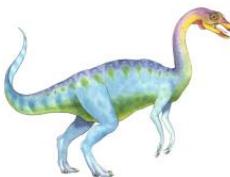


# Peterson's Solution

---

- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **int turn;**
  - **boolean flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section.
  - **flag[i] = true** implies that process  $P_i$  is ready!



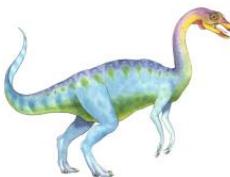


# Algorithm for Process $P_i$

```
while (true) {  
  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j)  
        ;  
  
    /* critical section */  
  
    flag[i] = false;  
  
    /* remainder section */  
  
}
```

$P_i$  wait and let  $P_j$  run



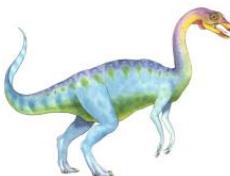


# Algorithm for Process $P_j$

```
while (true) {  
  
    flag[j] = true;  
    turn = i;  
    while (flag[i] && turn == i)  
        ;  
  
    /* critical section */  
  
    flag[j] = false;  
  
    /* remainder section */  
  
}
```

$P_j$  wait and let  $P_i$  run





# Correctness of Peterson's Solution

---

- Provable that the three CS requirement are met:

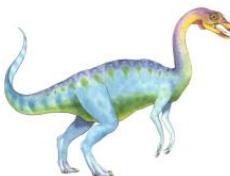
1. Mutual exclusion is preserved

$P_i$  enters CS only if:

either **flag[j] = false** or **turn = i**

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met



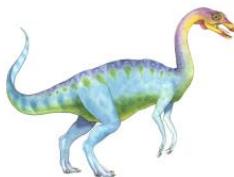


# Peterson's Solution and Modern Architecture

---

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!





# Modern Architecture Example

---

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

```
while (!flag)  
;  
print x
```

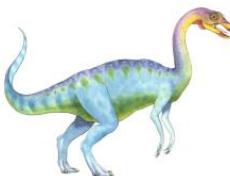
- Thread 2 performs

```
x = 100;  
flag = true
```

- What is the expected output?

100





# Modern Architecture Example (Cont.)

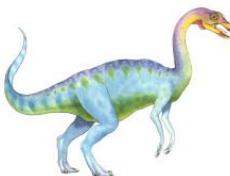
- However, since the variables `flag` and `x` are independent of each other, the instructions:

```
flag = true;  
x = 100;
```

for Thread 2 may be reordered

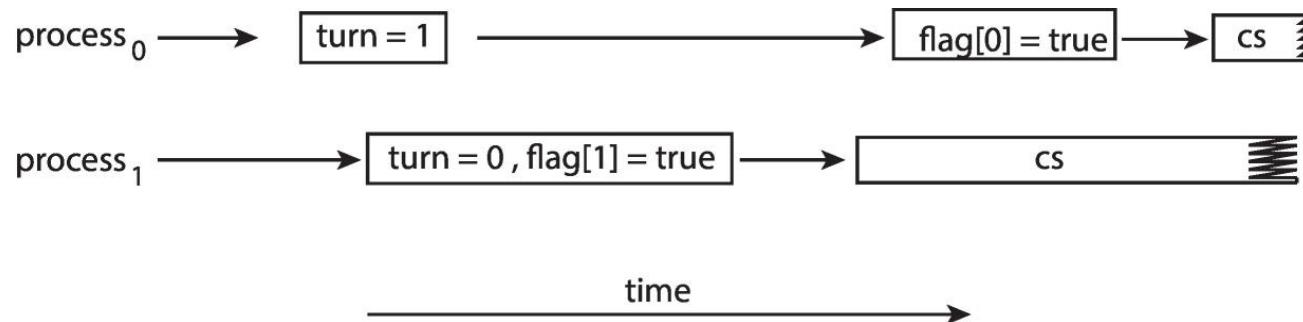
- If this occurs, the output may be 0!





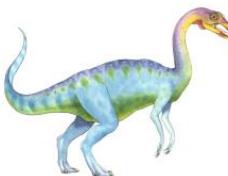
# Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



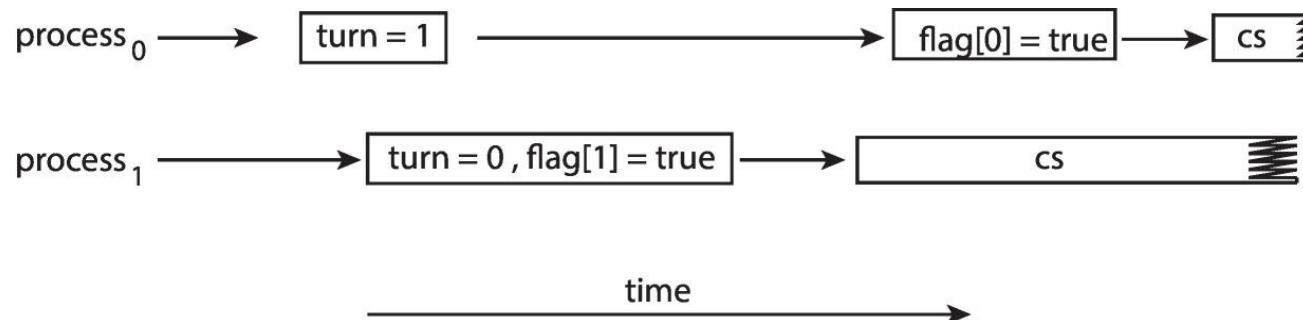
- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.





# Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



Pi:

```
turn =1  
flag[0]=true  
while(flag[1] and turn ==1);  
/* critical section */
```

**flag[0] = false;**

```
/* remainder section */  
turn = 0 can get in
```

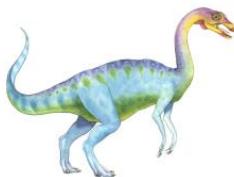
Pj:

```
turn =0  
flag[1]=true  
while(flag[0] and turn ==0);  
/* critical section */
```

**flag[1] = false;**

```
/* remainder section */  
flag[0] = false can get in
```





# Modern Architecture Example

---

- Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

- Thread 1 performs

```
while (!flag)  
;  
print x
```

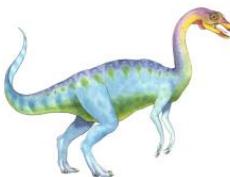
- Thread 2 performs

```
x = 100;  
flag = true
```

- What is the expected output?

100



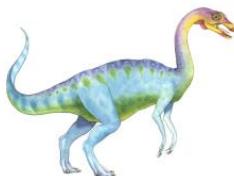


# Memory Barrier Instructions

---

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.





# Modern Architecture Example (Cont.)

---

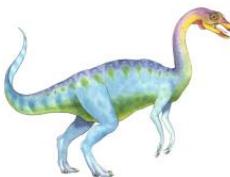
- However, since the variables `flag` and `x` are independent of each other, the instructions:

```
flag = true;  
x = 100;
```

for Thread 2 may be reordered

- If this occurs, the output may be 0!





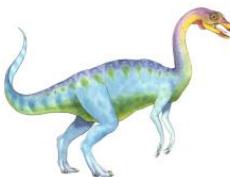
# Memory Barrier Example

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```
- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```
- For Thread 1 we are guaranteed that the value of flag is loaded before the value of x.
- For Thread 2 we ensure that the assignment to x occurs before the assignment flag.



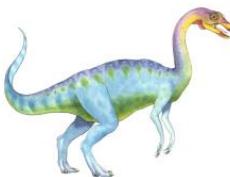


# Synchronization Hardware

---

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - ▶ Operating systems using this not broadly scalable
- We will look at three forms of hardware support:
  1. Hardware instructions
  2. Atomic variables



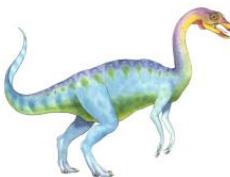


# Hardware Instructions

---

- **Special hardware instructions** that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction





# The test\_and\_set Instruction

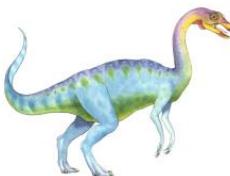
- Definition

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

- Properties

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**





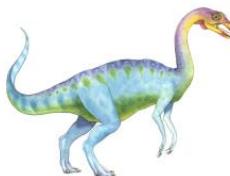
# Solution Using test\_and\_set()

- Shared boolean variable `lock`, initialized to `false`
- Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
    /* remainder section */  
} while (true);
```

- Does it solve the critical-section problem?
- First, false get into, then true, other can not get into





# The compare\_and\_swap Instruction

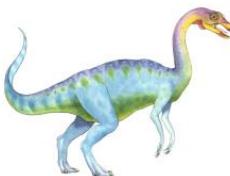
## ■ Definition

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

## ■ Properties

- Executed atomically
- Returns the original value of passed parameter **value**
- Set the variable **value** the value of the passed parameter **new\_value** but only if **\*value == expected** is true. That is, the swap takes place only under this condition.





# Solution using compare\_and\_swap

---

- Shared integer `lock` initialized to 0;
- Solution:

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

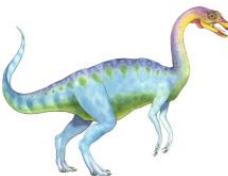
    /* critical section */

    lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?



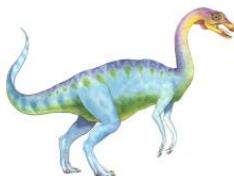


# Mutex Locks

---

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock(互斥锁)
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **acquire()** a lock
  - Then **release()** the lock
- Calls to **acquire()** and **release()** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**(忙等待)
  - This lock therefore called a **spinlock**(自旋锁)



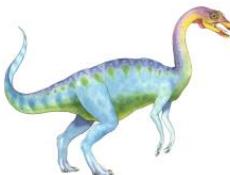


# Solution to CS Problem Using Mutex Locks

```
while (true) {
    acquire lock
    critical section
    release lock
    remainder section
}

while(true)
It is still busy waiting
```





# Semaphore(信号量)

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()** 2个函数,
    - ▶ Originally called **P()** and **V()**

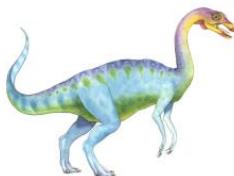
- Definition of the **wait()** operation

```
wait(S) { //wait函数  
    while (S <= 0)//S 是整型变量  
        ; // busy wait  
    S--;  
}
```

- Definition of the **signal()** operation

```
signal(S) {  
    S++;  
}
```



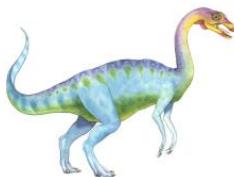


# Semaphore (Cont.)

---

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore **S** as a binary semaphore
- With semaphores we can solve various synchronization problems





# Semaphore Usage Example

---

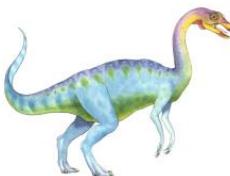
- Solution to the CS Problem
  - Create a semaphore “**mutex**” initialized to 1

```
    wait(mutex);
```

CS

```
    signal(mutex);
```



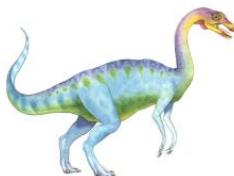


# Semaphore Implementation

---

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution



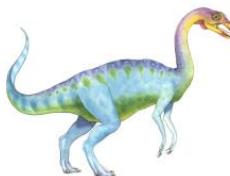


# Semaphore Implementation with no Busy waiting

---

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **Block**(阻塞) – place the process invoking the operation on the appropriate waiting queue
  - **Wakeup**(唤醒) – remove one of processes in the waiting queue and place it in the ready queue



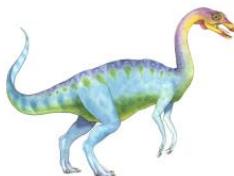


# Implementation with no Busy waiting (Cont.)

- Waiting queue

```
typedef struct {  
    int value;  
    struct process *list;  
} semaphore;
```

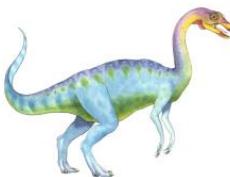




# Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) { //P operation
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
//no while
signal(semaphore *S) { //V operation
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```





# Problems with Semaphores

---

- Incorrect use of semaphore operations:
  - `signal(mutex)`   ...   `wait(mutex)`
  - `wait(mutex)`   ...   `wait(mutex)`
  - Omitting of `wait (mutex)` and/or `signal (mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.





# Monitors 管程

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- Pseudocode syntax of a monitor:

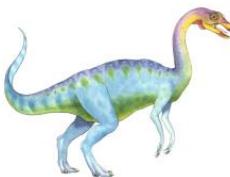
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }

    procedure P2 (...) { ... }

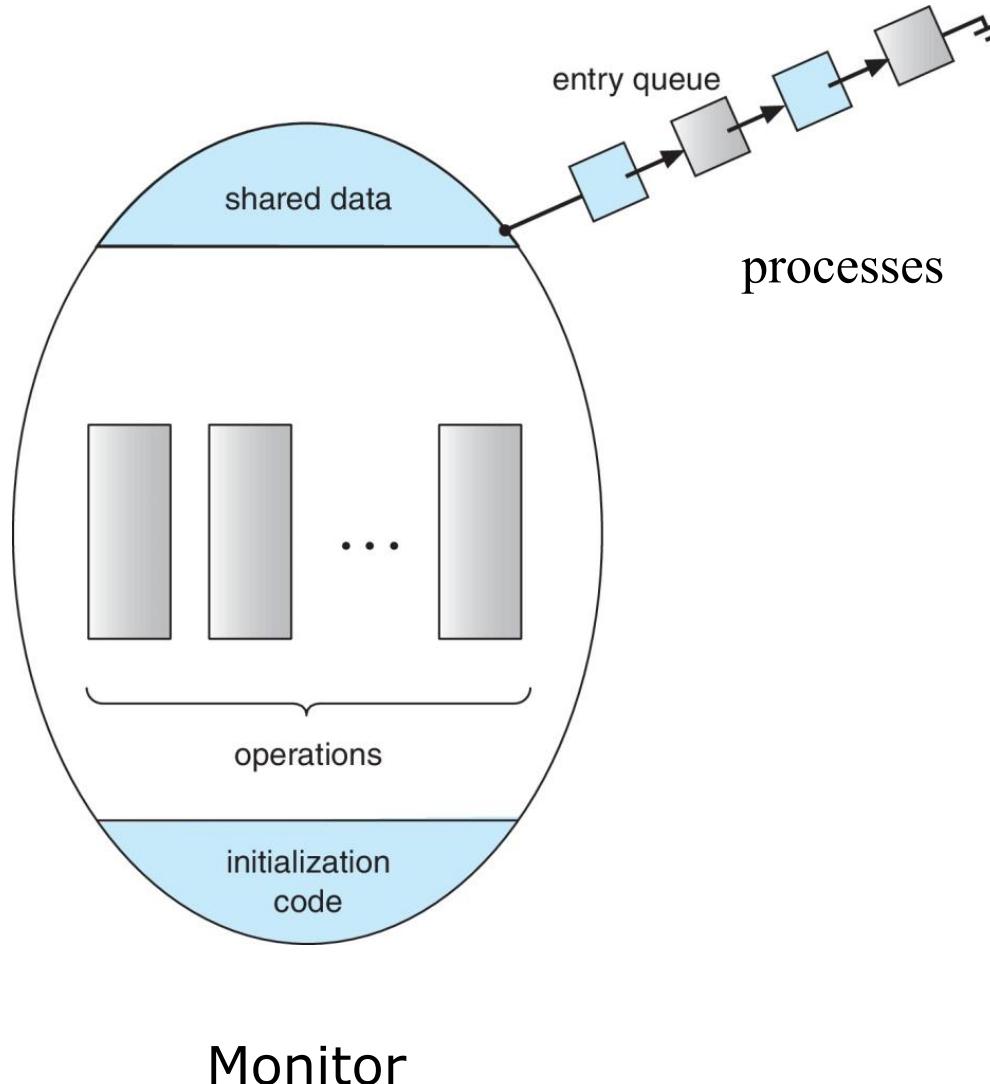
    procedure Pn (...) { .....}

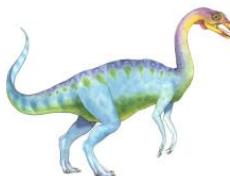
    initialization code (...) { ... }
}
```





# Schematic view of a Monitor





# Monitor Implementation Using Semaphores

---

- Variables

```
semaphore mutex  
mutex = 1
```

- Each procedure  $P$  is replaced by

```
wait(mutex);  
...  
body of P;  
...  
signal(mutex);
```

- Mutual exclusion within a monitor is ensured





# Monitor Implementation Using Semaphores

- Variables

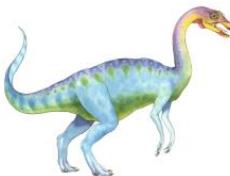
```
semaphore mutex; // (initially = 1)
semaphore next; // (initially = 0)
int next_count = 0; // number of processes waiting
                    inside the monitor
```

- Each function  $P$  will be replaced by

```
wait(mutex);
...
body of P;
...
if (next_count > 0)
    signal(next) // release to next
else
    signal(mutex); // release mutex
```

- Mutual exclusion within a monitor is ensured





# Java Synchronization

---

- Java provides rich set of synchronization features:
  - Java monitors
  - Reentrant locks
  - Semaphores
  - Condition variables



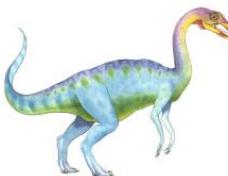


# Java Monitors

---

- Every Java object has associated with it a single lock.
- If a method is declared as **synchronized**, a calling thread must own the lock for the object.
- If the lock is owned by another thread, the calling thread must wait for the lock until it is released.
- Locks are released when the owning thread exits the **synchronized** method.





# Bounded Buffer – Java Synchronization

```
public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

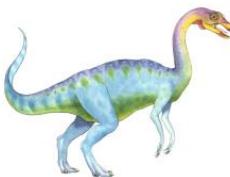
    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

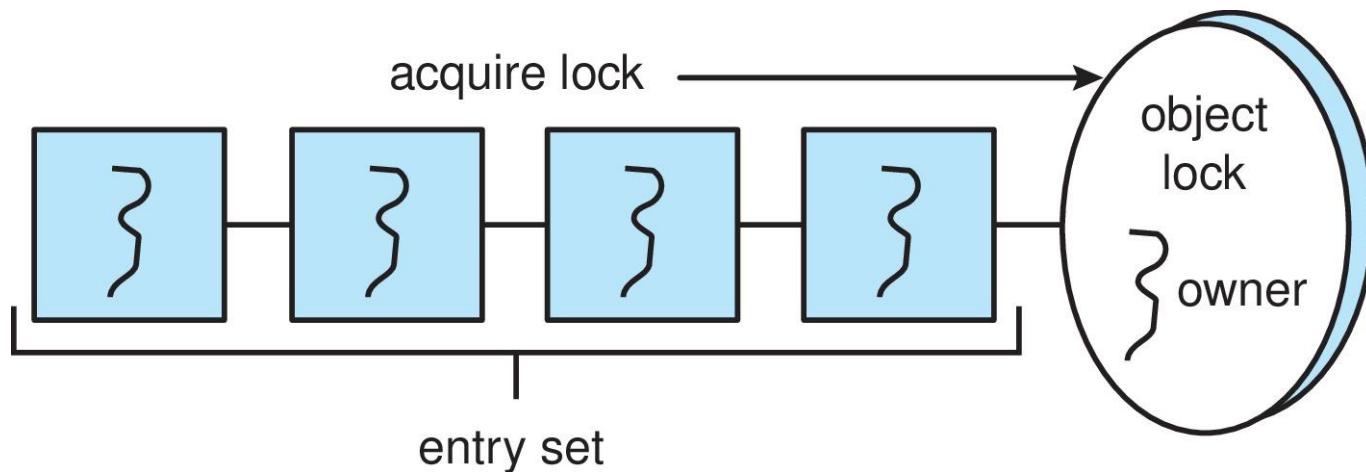
    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}
```

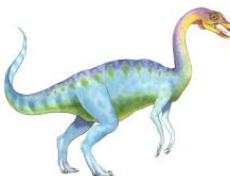




# Java Synchronization

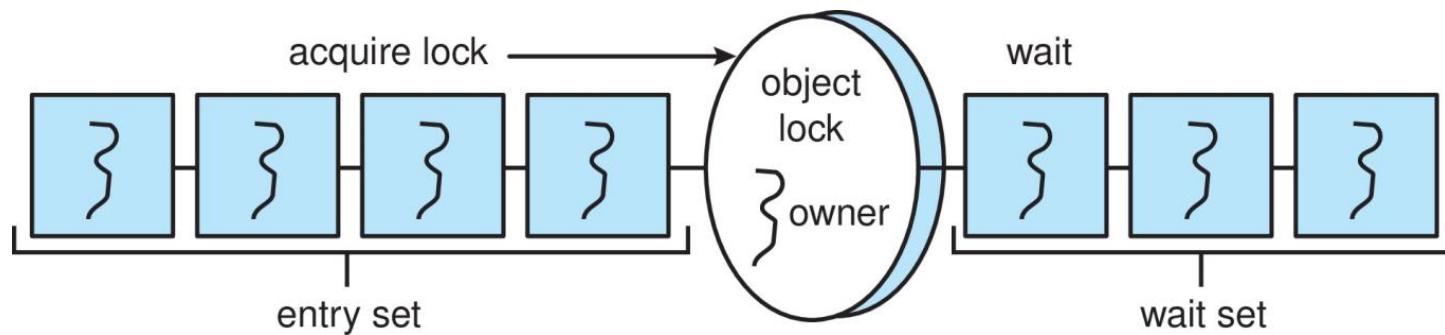
- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**: 入口集合

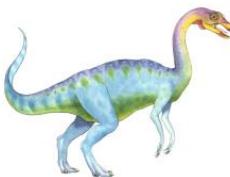




# Java Synchronization

- Similarly, each object also has a **wait set**. 等待集合
- When a thread calls **wait()**:
  1. It releases the lock for the object
  2. The state of the thread is set to blocked
  3. The thread is placed in the wait set for the object



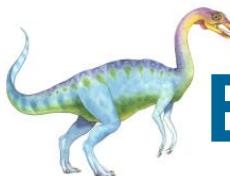


# Java Synchronization

---

- A thread typically calls `wait()` when it is waiting for a condition to become true.
- How does a thread get notified?
- When a thread calls `notify()`:
  1. An arbitrary thread T is selected from the wait set
  2. T is moved from the wait set to the entry set
  3. Set the state of T from blocked to runnable.
- T can now compete for the lock to check if the condition it was waiting for is now true.





# Bounded Buffer – Java Synchronization

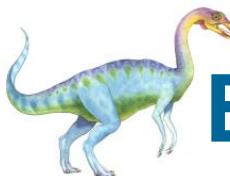
---

```
/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;

    notify();
}
```





# Bounded Buffer – Java Synchronization

```
/* Consumers call this method */
public synchronized E remove() {
    E item;

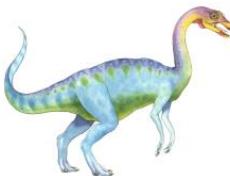
    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    notify();

    return item;
}
```





# Java Reentrant Locks

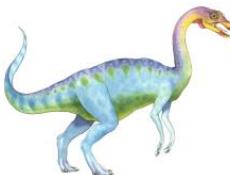
---

- Similar to mutex locks
- The **finally** clause ensures the lock will be released in case an exception occurs in the **try** block.

```
Lock key = new ReentrantLock();

key.lock();
try {
    /* critical section */
}
finally {
    key.unlock();
}
```





# Java Semaphores

---

- Constructor:

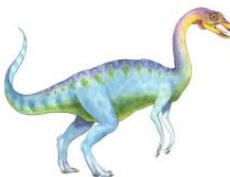
```
Semaphore(int value);
```

- Usage:

```
Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```





# Java Condition Variables

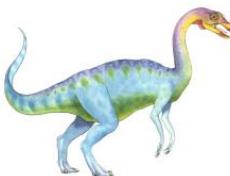
---

- Condition variables are associated with an **ReentrantLock**.
- Creating a condition variable using **newCondition()** method of **ReentrantLock**:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- A thread waits by calling the **await()** method, and signals by calling the **signal()** method.





# Java Condition Variables

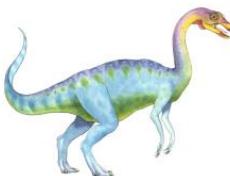
---

- Example:
- Five threads numbered 0 .. 4
- Shared variable **turn** indicating which thread's turn it is.
- Thread calls **doWork()** when it wishes to do some work. (But it may only do work if it is their turn.)
- If not their turn, wait
- If their turn, do some work for awhile .....
- When completed, notify the thread whose turn is next.
- Necessary data structures:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
    condVars[i] = lock.newCondition();
```





# Java Condition Variables

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
         */
        if (threadNumber != turn)
            condVars[threadNumber].await();
    }

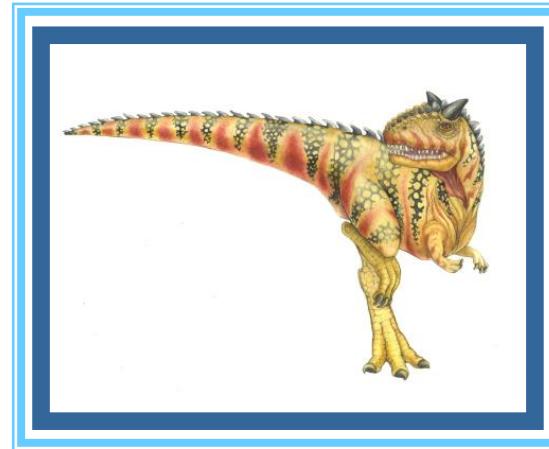
    /**
     * Do some work for awhile ...
     */

    /**
     * Now signal to the next thread.
     */
    turn = (turn + 1) % 5;
    condVars[turn].signal();
}

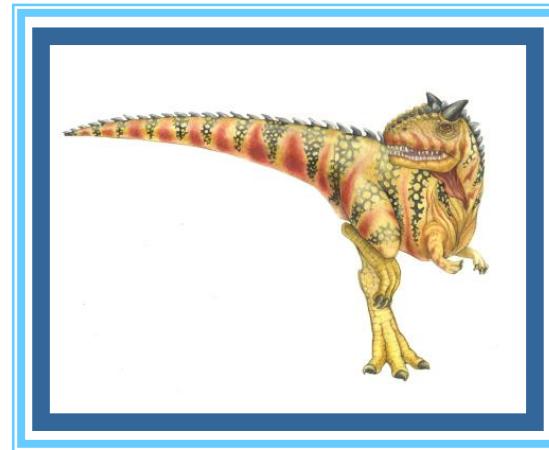
catch (InterruptedException ie) { }
finally {
    lock.unlock();
}
}
```

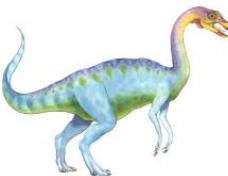


# End of Chapter 6



# Chapter 8: Deadlocks



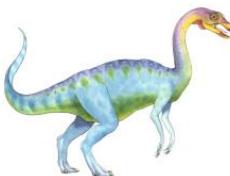


# Outline

---

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock





# Chapter Objectives

---

- Illustrate how deadlock can occur when mutex locks are used
- Define the four necessary conditions that characterize deadlock
- Identify a deadlock situation in a resource allocation graph
- Evaluate the four different approaches for preventing deadlocks
- Apply the banker's algorithm for deadlock avoidance
- Apply the deadlock detection algorithm
- Evaluate approaches for recovering from deadlock



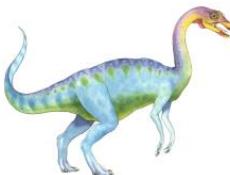


# System Model

---

- System consists of resources
- Resource types  $R_1, R_2, \dots, R_m$ 
  - *CPU cycles, memory space, I/O devices*
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**



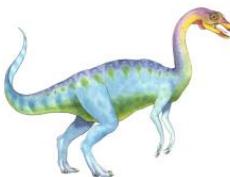


# Deadlock with Semaphores

---

- Data:
  - A semaphore  $s_1$  initialized to 1, resource
  - A semaphore  $s_2$  initialized to 1, resource
- Two threads  $T_1$  and  $T_2$
- $T_1$ :
  - `wait( $s_1$ )`
  - `wait( $s_2$ )`
- $T_2$ :
  - `wait( $s_2$ )`
  - `wait( $s_1$ )`





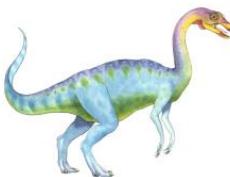
# Deadlock Characterization

---

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one thread at a time can use a resource
- **Hold and wait:** a thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption:** a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
- **Circular wait:** there exists a set  $\{T_0, T_1, \dots, T_n\}$  of waiting threads such that  $T_0$  is waiting for a resource that is held by  $T_1$ ,  $T_1$  is waiting for a resource that is held by  $T_2, \dots, T_{n-1}$  is waiting for a resource that is held by  $T_n$ , and  $T_n$  is waiting for a resource that is held by  $T_0$ .



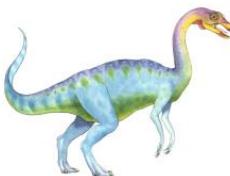


# Resource-Allocation Graph

A set of vertices  $V$  and a set of edges  $E$ .

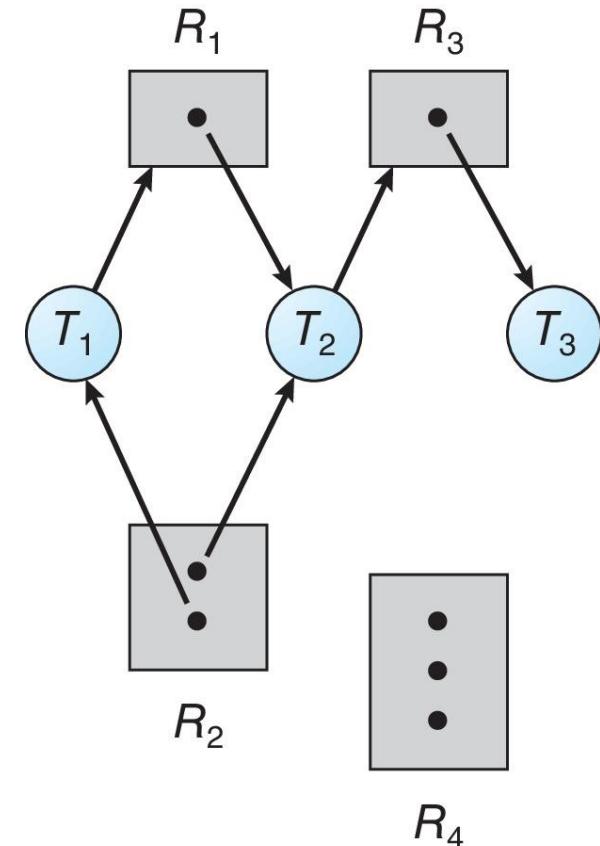
- $V$  is partitioned into two types:
  - $T = \{T_1, T_2, \dots, T_n\}$ , the set consisting of all the threads in the system.
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed edge  $T_i \rightarrow R_j$
- **assignment edge** – directed edge  $R_j \rightarrow T_i$

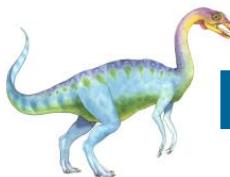




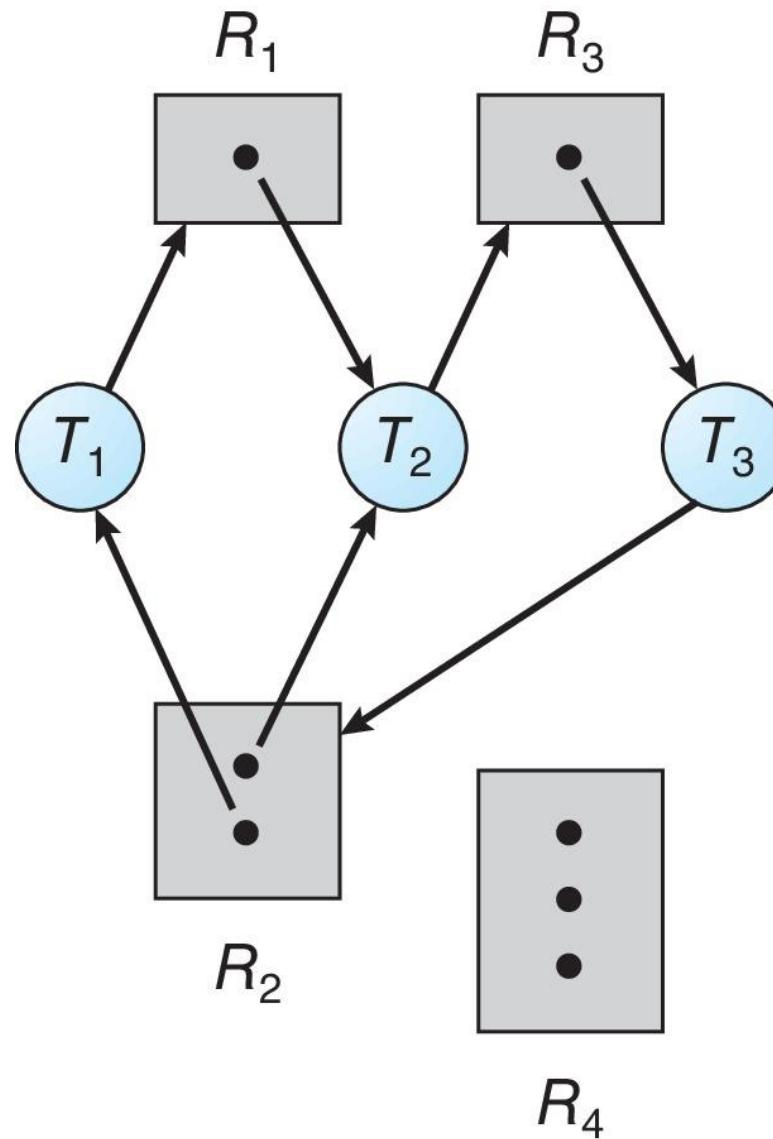
# Resource Allocation Graph Example

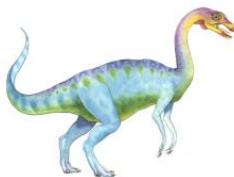
- One instance of  $R_1$
- Two instances of  $R_2$
- One instance of  $R_3$
- Three instances of  $R_4$
- $T_1$  holds one instance of  $R_2$  and is waiting for an instance of  $R_1$
- $T_2$  holds one instance of  $R_1$ , one instance of  $R_2$ , and is waiting for an instance of  $R_3$
- $T_3$  holds one instance of  $R_3$



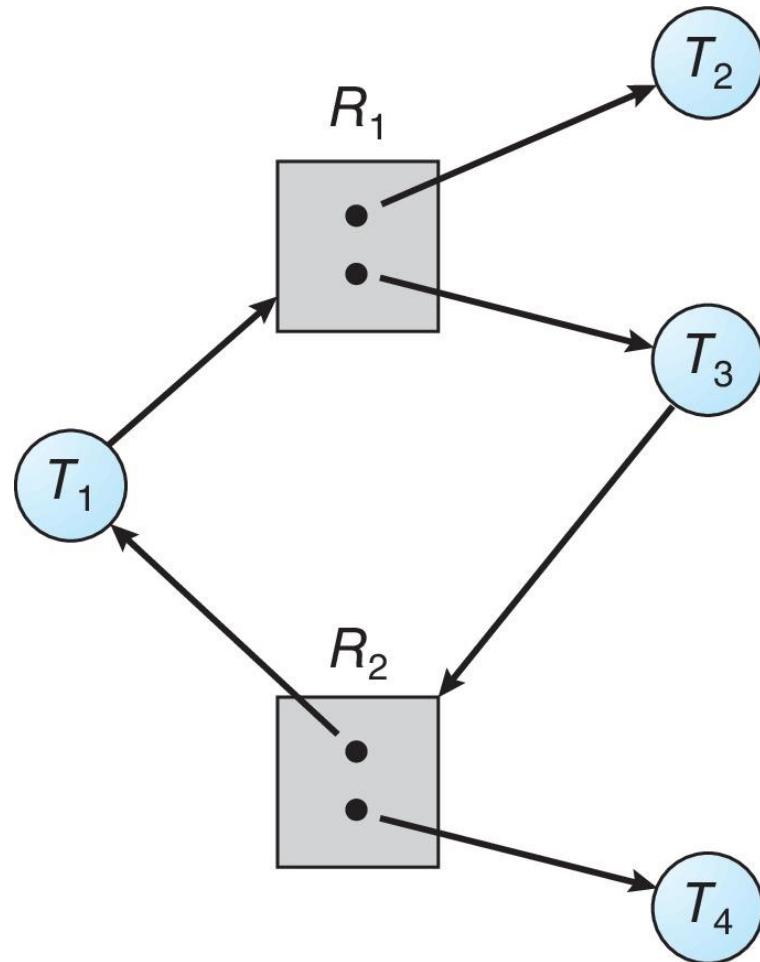


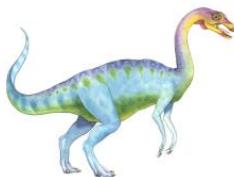
# Resource Allocation Graph with a Deadlock





# Graph with a Cycle But no Deadlock



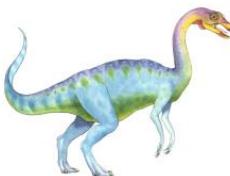


# Basic Facts

---

- If graph contains no cycles  $\Rightarrow$  no deadlock
- If graph contains a cycle  $\Rightarrow$ 
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock



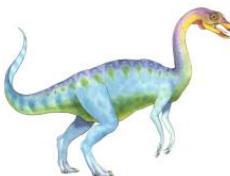


# Methods for Handling Deadlocks

---

- Ensure that the system will **never** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.





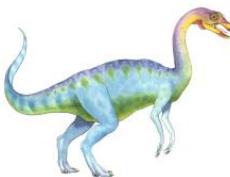
# Deadlock Prevention

---

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a thread requests a resource, it does not hold any other resources
  - Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it.
  - Low resource utilization; starvation possible





# Deadlock Prevention (Cont.)

---

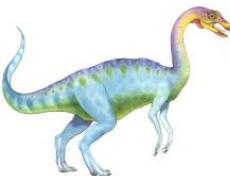
- **No Preemption:**

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
- Preempted resources are added to the list of resources for which the thread is waiting
- Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- **Circular Wait:**

- Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration



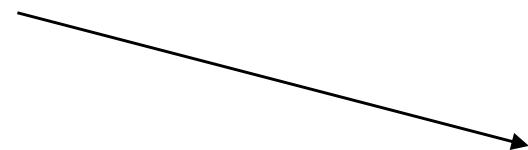


# Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:

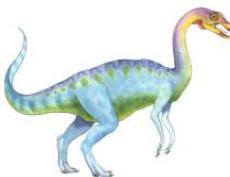
```
first_mutex = 1  
second_mutex = 5
```

code for **thread\_two** could not be written as follows:



```
/* thread_one runs in this function */  
void *do_work_one(void *param)  
{  
    pthread_mutex_lock(&first_mutex);  
    pthread_mutex_lock(&second_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&second_mutex);  
    pthread_mutex_unlock(&first_mutex);  
  
    pthread_exit(0);  
}  
  
/* thread_two runs in this function */  
void *do_work_two(void *param)  
{  
    pthread_mutex_lock(&second_mutex);  
    pthread_mutex_lock(&first_mutex);  
    /**  
     * Do some work  
     */  
    pthread_mutex_unlock(&first_mutex);  
    pthread_mutex_unlock(&second_mutex);  
  
    pthread_exit(0);  
}
```





# Deadlock Avoidance

---

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each thread declare the **maximum number** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes



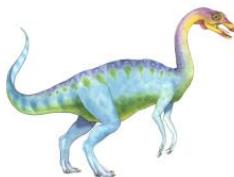


# Safe State

---

- When a thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence  $\langle T_1, T_2, \dots, T_n \rangle$  of ALL the threads in the systems such that for each  $T_i$ , the resources that  $T_i$  can still request can be satisfied by currently available resources + resources held by all the  $T_j$ , with  $j < i$
- That is:
  - If  $T_i$  resource needs are not immediately available, then  $T_i$  can wait until all  $T_j$  have finished
  - When  $T_j$  is finished,  $T_i$  can obtain needed resources, execute, return allocated resources, and terminate
  - When  $T_i$  terminates,  $T_{i+1}$  can obtain its needed resources, and so on



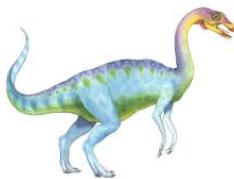


# Basic Facts

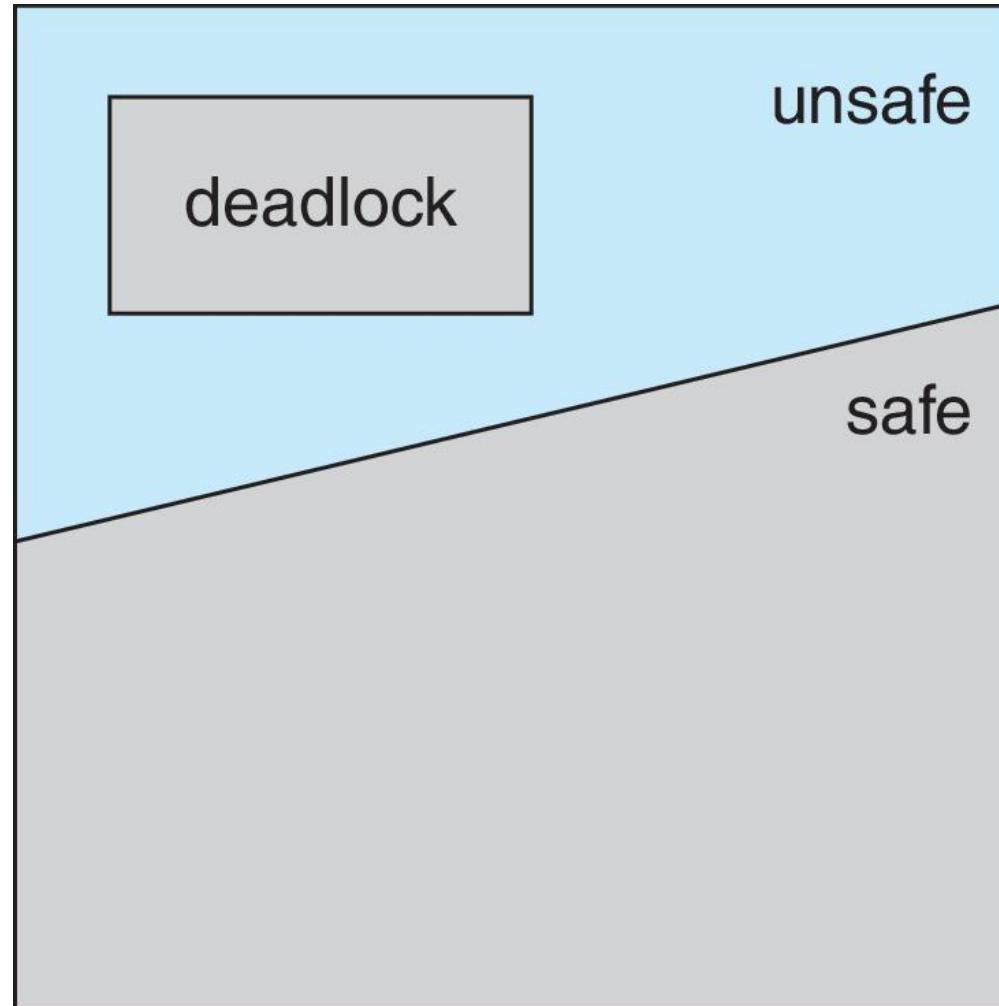
---

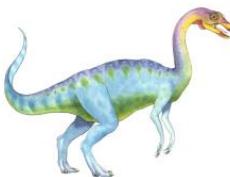
- If a system is in safe state  $\Rightarrow$  no deadlocks
- If a system is in unsafe state  $\Rightarrow$  possibility of deadlock
- Avoidance  $\Rightarrow$  ensure that a system will never enter an unsafe state.





# Safe, Unsafe, Deadlock State



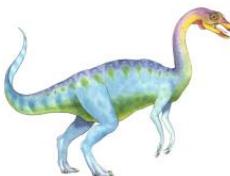


# Avoidance Algorithms

---

- Single instance of a resource type
  - Use a resource-allocation graph
  
- Multiple instances of a resource type
  - Use the Banker's Algorithm





# Resource-Allocation Graph Scheme

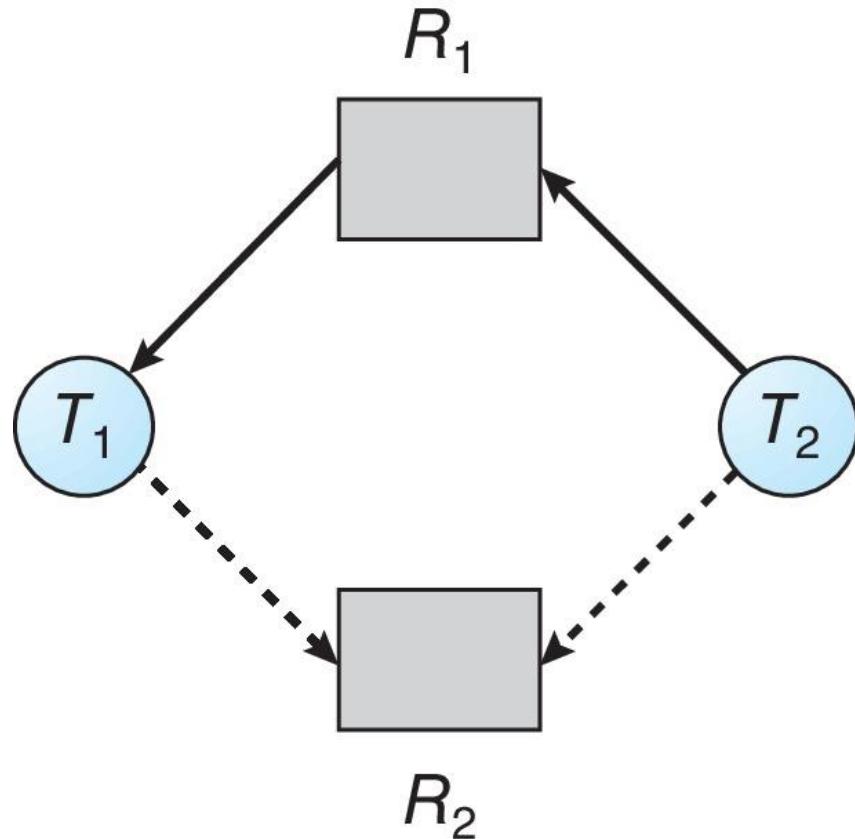
---

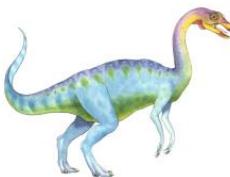
- **Claim edge(需求边)**  $T_i \rightarrow R_j$  indicated that process  $T_j$  may request resource  $R_j$ ; represented by a dashed line
- Claim edge converts to request edge when a thread requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the thread
- When a resource is released by a thread, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



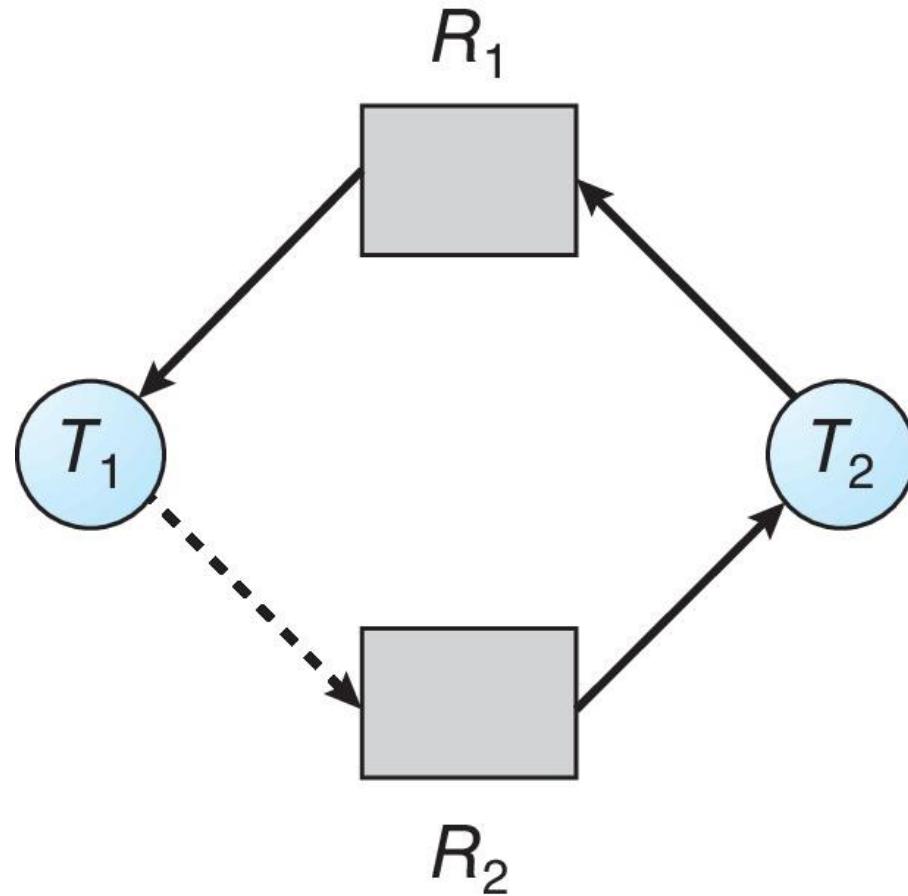


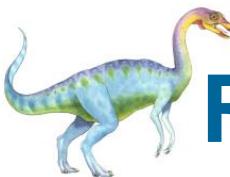
# Resource-Allocation Graph





# Unsafe State In Resource-Allocation Graph



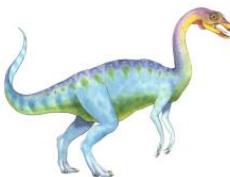


# Resource-Allocation Graph Algorithm

---

- Suppose that thread  $T_i$  requests a resource  $R_j$
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph



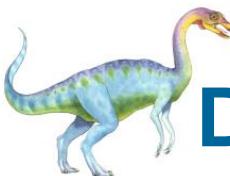


# Banker's Algorithm

---

- Multiple instances of resources
- Each thread must have a priori claim maximum use
- When a thread requests a resource, it may have to wait
- When a thread gets all its resources it must return them in a finite amount of time





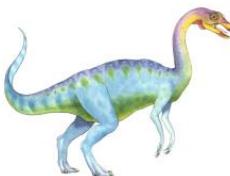
# Data Structures for the Banker's Algorithm

Let  $n$  = number of processes, and  $m$  = number of resources types.

- **Available:** Vector of length  $m$ . If  $\text{available}[j] = k$ , there are  $k$  instances of resource type  $R_j$  available
- **Max:**  $n \times m$  matrix. If  $\text{Max}[i,j] = k$ , then process  $T_i$  may request at most  $k$  instances of resource type  $R_j$
- **Allocation:**  $n \times m$  matrix. If  $\text{Allocation}[i,j] = k$  then  $T_i$  is currently allocated  $k$  instances of  $R_j$
- **Need:**  $n \times m$  matrix. If  $\text{Need}[i,j] = k$ , then  $T_i$  may need  $k$  more instances of  $R_j$  to complete its task

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$





# Safety Algorithm

---

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively.  
Initialize:

**Work = Available**

**Finish [i] = false for  $i = 0, 1, \dots, n-1$**

2. Find an  $i$  such that both:

(a) **Finish [i] = false**

(b) **Need<sub>i</sub> ≤ Work**

If no such  $i$  exists, go to step 4

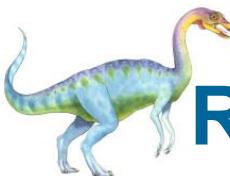
3. **Work = Work + Allocation<sub>i</sub>**,

**Finish[i] = true**

go to step 2

4. If **Finish [i] == true** for all  $i$ , then the system is in a safe state





# Resource-Request Algorithm for Process $P_i$

$\text{Request}_i$  = request vector for process  $T_i$ . If  $\text{Request}_i[j] = k$  then process  $T_i$  wants  $k$  instances of resource type  $R_j$

1. If  $\text{Request}_i \leq \text{Need}_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If  $\text{Request}_i \leq \text{Available}$ , go to step 3. Otherwise  $T_i$  must wait, since resources are not available
3. Pretend to allocate requested resources to  $T_i$  by modifying the state as follows:

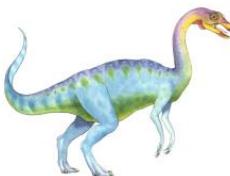
$$\text{Available} = \text{Available} - \text{Request}_i;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

- If safe  $\Rightarrow$  the resources are allocated to  $T_i$
- If unsafe  $\Rightarrow T_i$  must wait, and the old resource-allocation state is restored



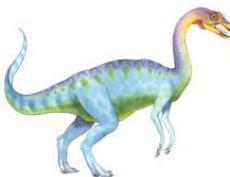


# Example of Banker's Algorithm

- 5 threads  $T_0$  through  $T_4$ ;  
3 resource types:  
 $A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)
- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>	<u>Available</u>	
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$T_0$	0	1	0	7	5	3
$T_1$	2	0	0	3	2	2
$T_2$	3	0	2	9	0	2
$T_3$	2	1	1	2	2	2
$T_4$	0	0	2	4	3	3





## Example (Cont.)

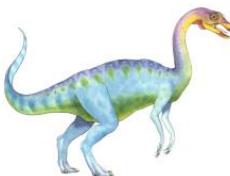
---

- The content of the matrix **Need** is defined to be **Max – Allocation**

	<u>Need</u>		
	A	B	C
$T_0$	7	4	3
$T_1$	1	2	2
$T_2$	6	0	0
$T_3$	0	1	1
$T_4$	4	3	1

- The system is in a safe state since the sequence  $\langle T_1, T_3, T_4, T_2, T_0 \rangle$  satisfies safety criteria





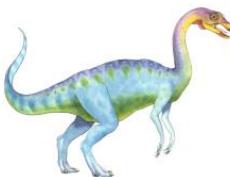
## Example: $P_1$ Request (1,0,2)

- Check that Request  $\leq$  Available (that is,  $(1,0,2) \leq (3,3,2) \Rightarrow$  true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
$T_0$	0 1 0	7 4 3	2 3 0
$T_1$	3 0 2	0 2 0	
$T_2$	3 0 2	6 0 0	
$T_3$	2 1 1	0 1 1	
$T_4$	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence  $< T_1, T_3, T_4, T_0, T_2 >$  satisfies safety requirement



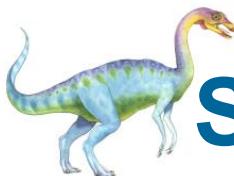


# Deadlock Detection

---

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

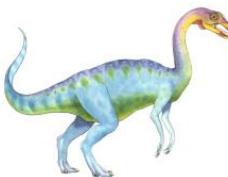




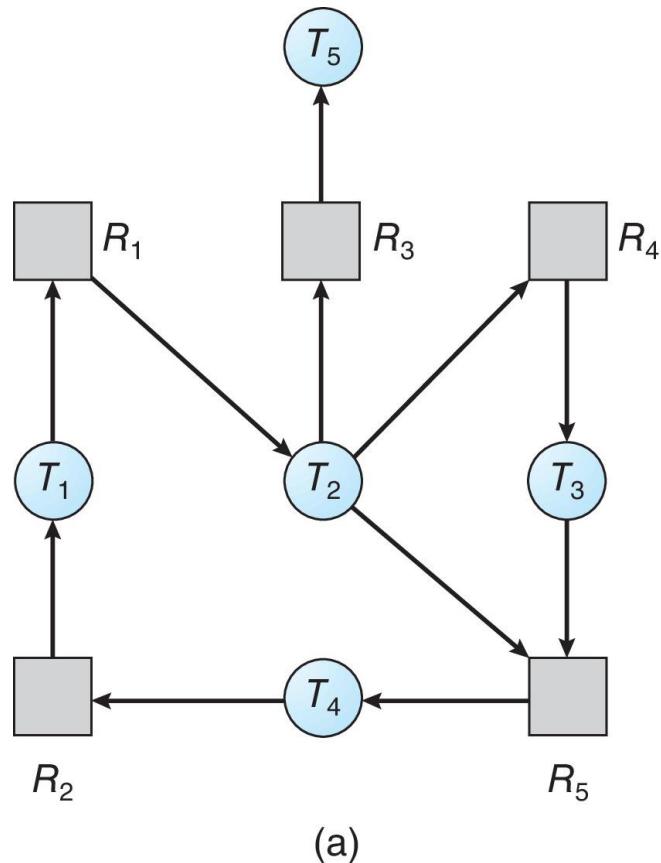
# Single Instance of Each Resource Type

- Maintain **wait-for** graph
  - Nodes are threads
  - $T_i \rightarrow T_j$  if  $T_i$  is waiting for  $T_j$
- Periodically invoke an algorithm that searches for a cycle in the graph.  
If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of  $n^2$  operations, where  $n$  is the number of vertices in the graph

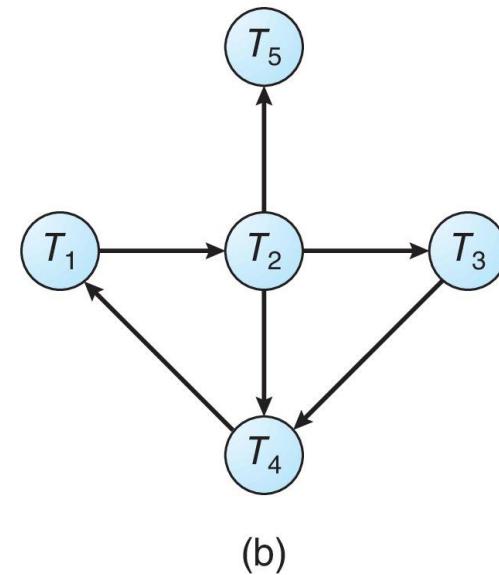




# Resource-Allocation Graph and Wait-for Graph



(a)

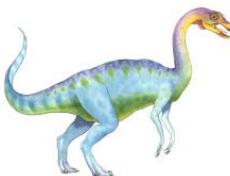


(b)

Resource-Allocation Graph

Corresponding wait-for graph



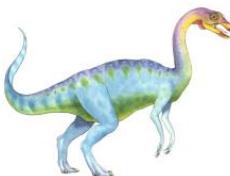


# Several Instances of a Resource Type

---

- **Available:** A vector of length  $m$  indicates the number of available resources of each type
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each thread.
- **Request:** An  $n \times m$  matrix indicates the current request of each thread. If  $\text{Request}[i][j] = k$ , then thread  $T_i$  is requesting  $k$  more instances of resource type  $R_j$ .



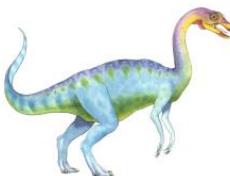


# Detection Algorithm

1. Let **Work** and **Finish** be vectors of length  $m$  and  $n$ , respectively  
Initialize:
  - a) **Work = Available**
  - b) For  $i = 1, 2, \dots, n$ , if **Allocation<sub>i</sub>**  $\neq 0$ , then  
**Finish[i] = false**; otherwise, **Finish[i] = true**
2. Find an index  $i$  such that both:
  - a) **Finish[i] == false**
  - b) **Request<sub>i</sub> ≤ Work**

If no such  $i$  exists, go to step 4





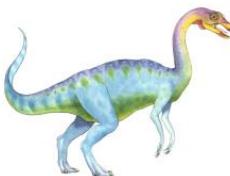
# Detection Algorithm (Cont.)

---

3.  $\text{Work} = \text{Work} + \text{Allocation}_i$ ,  
 $\text{Finish}[i] = \text{true}$   
go to step 2
4. If  $\text{Finish}[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $T_i$  is deadlocked

Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state





# Example of Detection Algorithm

- Five threads  $T_0$  through  $T_4$ ; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
$T_0$	0	1	0	0	0	0	0	0	0
$T_1$	2	0	0	2	0	2			
$T_2$	3	0	3	0	0	0			
$T_3$	2	1	1	1	0	0			
$T_4$	0	0	2	0	0	2			

- Sequence  $\langle T_0, T_2, T_3, T_1, T_4 \rangle$  will result in  $Finish[i] = true$  for all  $i$





# Example (Cont.)

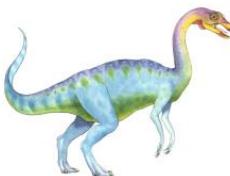
- $T_2$  requests an additional instance of type **C**

Request

	A	B	C
$T_0$	0	0	0
$T_1$	2	0	2
$T_2$	0	0	1
$T_3$	1	0	0
$T_4$	0	0	2

- State of system?
  - Can reclaim resources held by thread  $T_0$ , but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes  $T_1$ ,  $T_2$ ,  $T_3$ , and  $T_4$

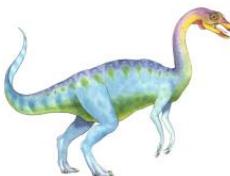




# Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - ▶ one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked threads “caused” the deadlock.

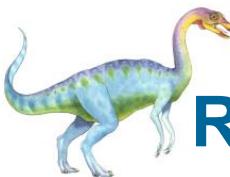




# Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the thread
  2. How long has the thread computed, and how much longer to completion
  3. Resources that the thread has used
  4. Resources that the thread needs to complete
  5. How many threads will need to be terminated
  6. Is the thread interactive or batch?





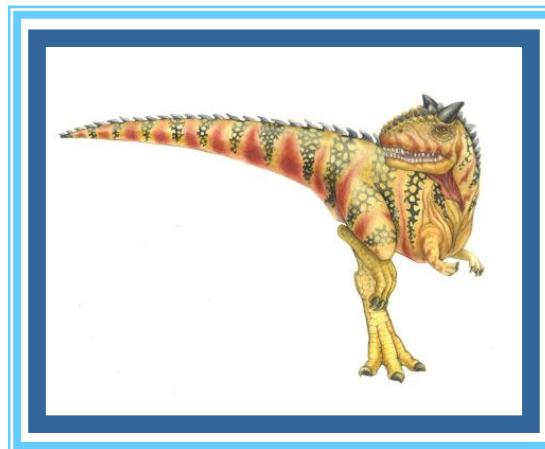
# Recovery from Deadlock: Resource Preemption

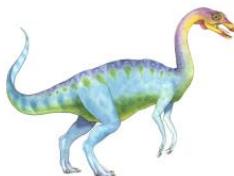
---

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart the thread for that state
- **Starvation** – same thread may always be picked as victim, include number of rollback in cost factor



# End of Chapter 8





## ■ 数字逻辑:

- 二进制, 补码、浮点数表示(了解)、布尔代数
- , - 逻辑门电路、数字逻辑系统设计、卡诺图化简、加法器、标志位
- 触发器, 计数器基本概念 (一般了解)
- 计算机系统基本结构, CPU: ALU + CU + Register, Memory, I/O以及总线
- I/O查询与中断基本概念

## ■ 汇编语言

- Register类型、数据类型定义、寻址方式、伪指令
- 汇编语言指令: 算术运算、逻辑运算、传输、条件、循环、移位指令等
- 过程定义与调用, Irvine32基本库
- 汇编编程

## ■ 操作系统

- 进程、线程基本概念, PCB, Context, 进/线程状态及转换, 进程间IPC方式。
- CPU进程调度算法, FCFS, SJF, RR, Priority
- 线程同步原理与方法, Peterson、互斥锁、信号量等, 原理的伪代码
- 死锁预防和避免, 银行家算法
- 内存基本概念, 虚拟内存, 页/帧





- 90 分钟，3 大题 (75分)
- 简单计算
- 回答问题
- 分析说明
- 编程

