Concept of Operating Systems

- ☐ A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and a user

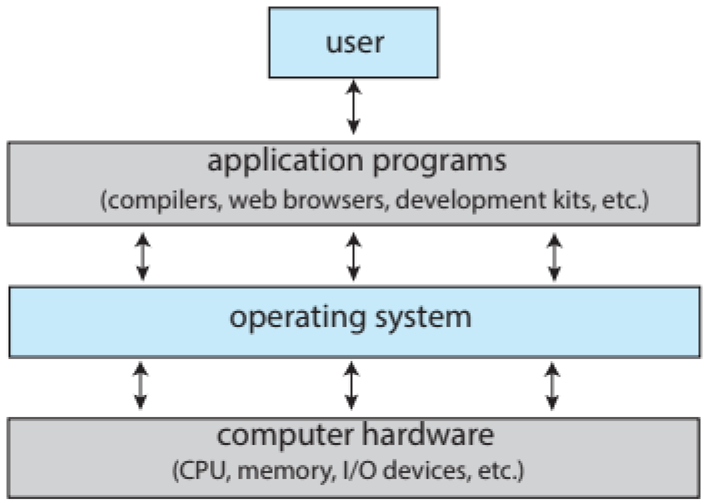


Figure 1.1 Abstract view of the components of a computer system.

- ☐
- ☐ The hardware— the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system.
- ☐ The application programs—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems.
- ☐ The operating system controls the hardware and coordinates its use among the various application programs for the various users.
- ☐ We next explore operating systems from two viewpoints: that of the user and that of the system.
- ☐ **User View**
  - ☐ The user's view of the computer varies according to the interface being used.
  - ☐ Many computer users sit with a laptop or in front of a PC consisting of a monitor, keyboard, and mouse.
  - ☐ In this case, the operating system is designed mostly for ease of use, with some attention paid to performance and security

and none paid to resource utilization—how various hardware and software resources are shared.
- ☐ Many users interact with mobile devices such as smartphones and tablets
- ☐ The user interface for mobile computers generally features a touch screen
- ☐ Many mobile devices also allow users to interact through a voice recognition interface, such as Apple's Siri.
- ☐ Some computers have little or no user view. For example, embedded computers in home devices and automobiles - designed primarily to run without user intervention.

- ☐ **System View**
  - ☐ In this context, we can view an operating system as a resource allocator.
  - ☐ A computer system has many resources that may be required to solve a problem: CPU time, memory space, storage space, I/O devices, and so on.
  - ☐ The operating system acts as the manager of these resources.
- ☐

Defining Operating Systems

- ☐ The fundamental goal of computer systems is to execute programs and to make solving user problems easier.
- ☐ Since bare hardware alone is not particularly easy to use, application programs are developed.
- ☐ These programs require certain common operations, such as those controlling the I/O devices.
- ☐ The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.
- ☐ operating system is the one program running at all times on the computer—usually called the **kernel**.
- ☐ Along with the kernel, there are two other types of programs: **system programs** and **application programs**

- ☐ System programs - associated with the operating system but are not necessarily part of the kernel
- ☐ Application programs - include all programs not associated with the operation of the system.
- ☐ Mobile operating systems often include not only a core kernel but also **middleware**—a set of software frameworks that provide additional services to application developers**.**
- ☐ An operating system (OS) functions as a mediator between application programs, utilities, users, and the computer hardware

BASIC ELEMENTS OF A COMPUTER:

- ☐ A computer consists of processor, memory, I/O components and system bus

- ☐ **Processor**: It Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the central processing unit
- ☐ **Main memory**: It Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. Main memory is also referred to as real memory or primary memory
- ☐ **I/O modules**: It moves data between the computer and its external environment. The external environment consists of a variety of devices, including secondary memory devices (e. g., disks), communications equipment, and terminals
- ☐ **System bus**: It provides the communication among processors, main memory, and I/O modules

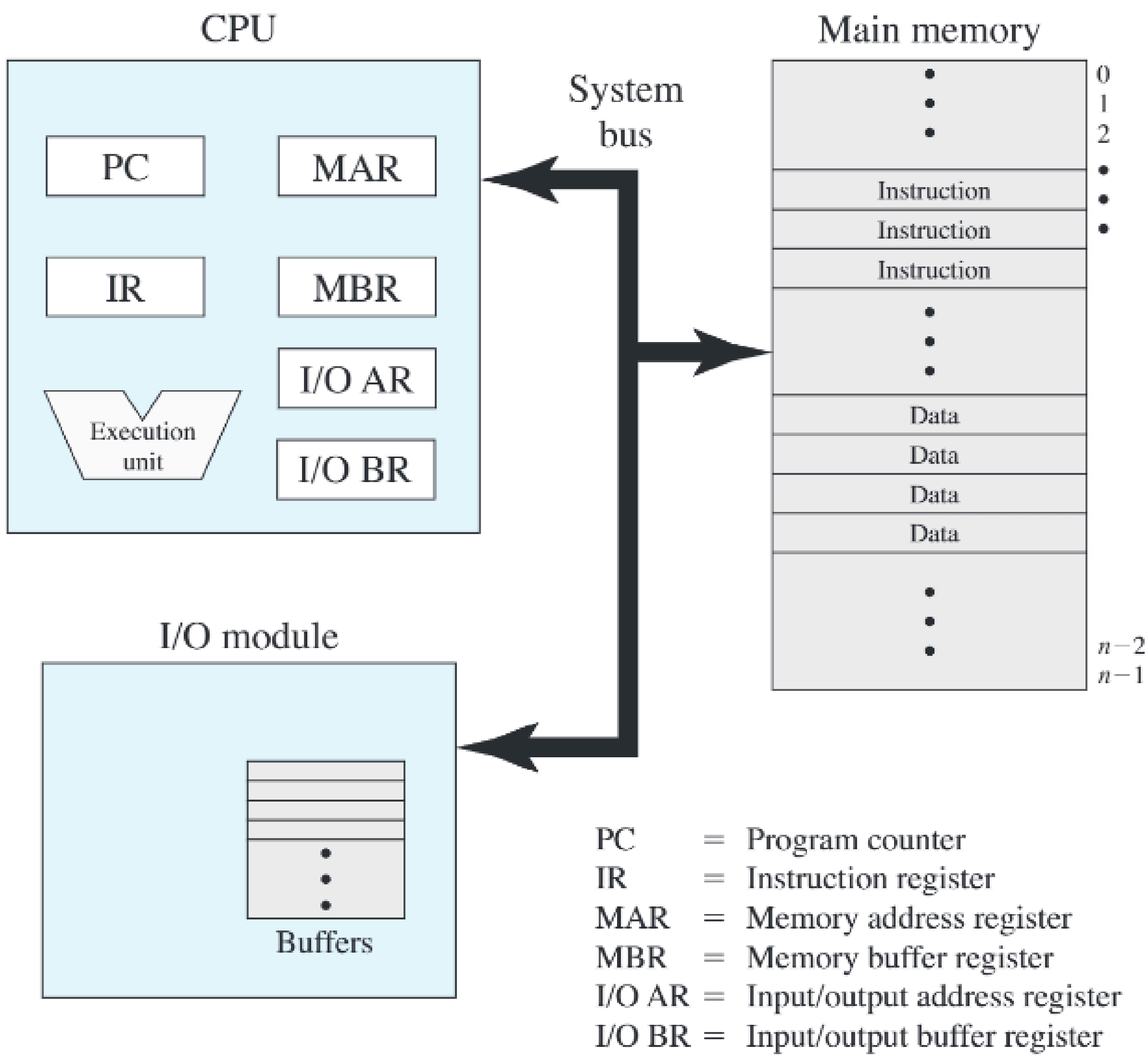


Figure 1.1 **Computer Components: Top-Level View**

- ☐ One of the processor's functions is to exchange data with memory. For this purpose, it typically makes use of two internal registers
- ☐ **A memory address registers (MAR),** which specifies the address in memory for the next read or write
- ☐ **A memory buffer register (MBR)**, which contains the data to be written into memory or which receives the data read from memory
- ☐ **An I/O address register (I/OAR)** specifies a particular I/O device
- ☐ **An I/O buffer register (I/OBR)** is used for the exchange of data between an I/O module and the processor

- ☐ **A memory module** consists of a set of locations, defined by sequentially numbered addresses.
- ☐ **An I/O module** transfers data from external devices to processor and memory, and vice versa. It contains internal buffers for temporarily holding data until they can be sent on
- ☐ **PROCESSOR REGISTERS** provide memory that is faster and smaller than main memory. Processor registers serve two functions
- ☐ **1.User-visible registers:**Enable the machine or assembly language programmer to minimize main memory references by optimizing register use
- ☐ A user-visible register is generally available to all programs, including application programs as well as system programs
- ☐ The types of User visible registers are
  - ☐ Data Registers - can be used with any machine instruction that performs operations on data
  - ☐ Address Registers - contain main memory addresses of data and instructions
- ☐ **2.Control and status register:**
  - ☐ A variety of processor registers are employed to control the operation of the processor
  - ☐ **MAR, MBR, IOAR, IOBR**
  - ☐ **Instruction register (IR):** It contains the instruction most recently fetched
  - ☐ **Program counter (PC):**Contains the address of the next instruction to be fetched

INSTRUCTION EXECUTION:

- ☐ A program to be executed by a processor consists of a set of instructions stored in Memory. The instruction processing consists of two steps
- ☐ The processor reads (fetches) instructions from memory one at a time **(fetch stage)**
- ☐ Execute the instruction.**(execute stage)**
- ☐ The processing required for a single instruction is called an **instruction cycle**

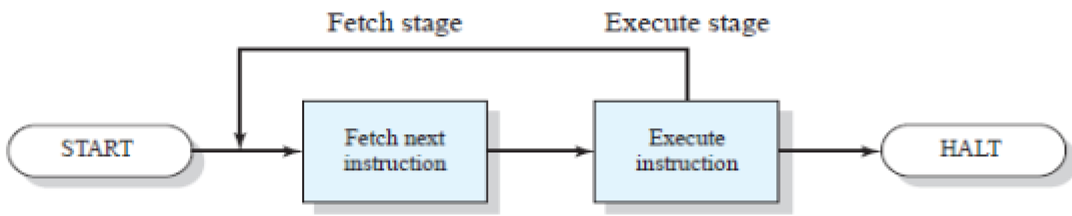
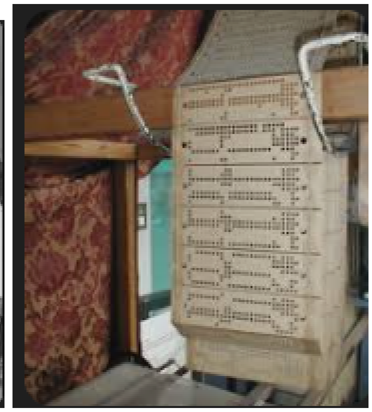


Figure 1.2 **Basic Instruction Cycle**

EVOLUTION OF OPERATING SYSTEM (or) Generations of OS (or) Types of OS

1. Serial processing [First Generation]

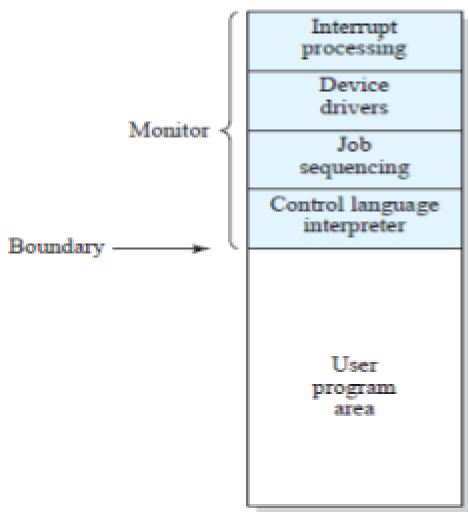
- During 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS
- Programs in machine code were loaded via the input device (e.g., a card reader)
- If an error halted the program, the error condition was indicated by the lights
- If the program proceeded to a normal completion, the output appeared on the printer
- These early systems presented two main problems:
- **1.Scheduling**
  - Most installations used a hardcopy sign-up sheet to reserve computer time
  - A user might sign up for an hour and finish in 45 minutes
  - On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem
- **2.Setup time**
  - A single program, called a **job**, could involve loading the compiler plus the high-level language program (source program) into memory
  - saving the compiled program (object program)
  - then loading and linking together the object program and common functions



2. Simple Batch Systems [Second Generation]

- In 1970’s
- The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor**
- the user **no longer has direct access to the processor**
- user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor
- Each program is constructed to branch back to the monitor when it completes processing, and the monitor automatically begins loading the next program

☐ 



Memory Layout for a Resident Monitor

☐

☐ The monitor controls the sequence of events. For this the monitor must always be in main memory and available for execution. That portion is referred to as the resident monitor

☐ The monitor reads in jobs one at a time from the input device .As it is read in, the current job is placed in the user program area, and control is passed to this job

☐ Once a job has been read in, the processor will encounter a branch instruction in the monitor that instructs the processor to continue execution at the start of the user program. The processor will then execute the instructions in the user program until it encounters an ending or error condition

☐ The monitor performs a scheduling function: A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time

☐ With each job, instructions are included in a form of job control language (JCL) which are denoted by the beginning $. This is a special type of programming language used to provide instructions to the monitor

☐ The overall format of the job is given as

```
$JOB
$FTN
•
•        FORTRAN instructions
•
$LOAD
$RUN
•
•        Data
•
$END
```

☐

☐ The memory protection leads to the concept of dual mode operation: **User mode** and **Kernel Mode**

☐

3. Multiprogrammed Batch Systems[Third Generation]

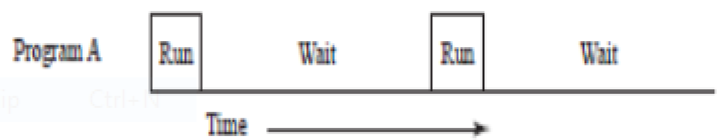
☐ Even in simple batch operating system, the processor is often idle. The problem is that I/O devices are slow compared to the processor

☐ Let us consider a program that processes a file of records and performs, on average, 100 machine instructions per record. The computer spends over 96% of its time waiting for I/O devices to finish transferring data to and from the file

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
Total	31 μs

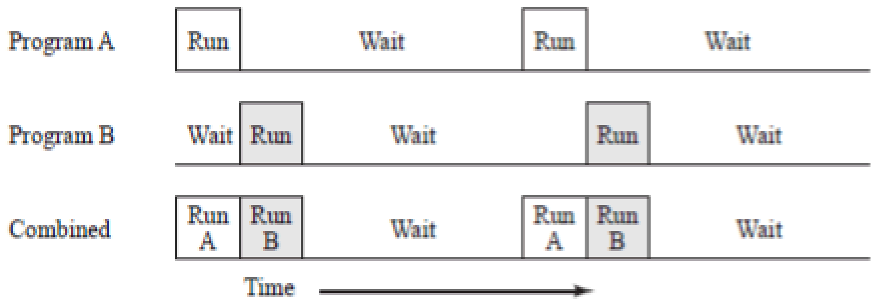
$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

- In uniprogramming we will have a single program in the main memory. The processor spends a certain amount of time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding. This inefficiency is not necessary

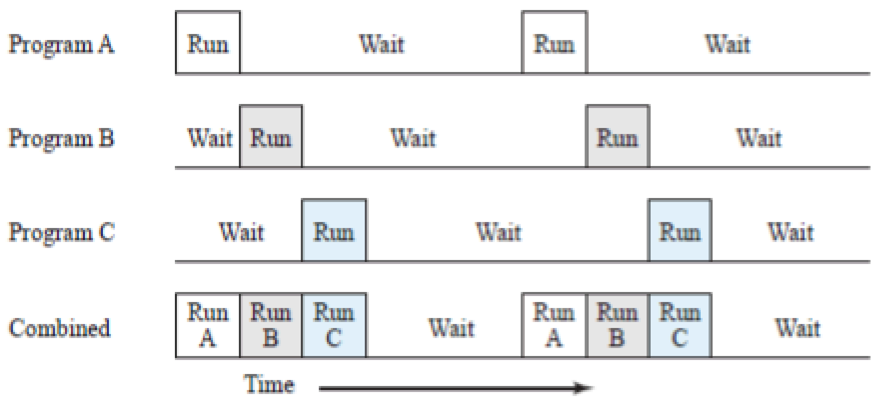


(a) Uniprogramming

- In Multiprogramming we will have an OS and more user programs. When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O.This approach is known as **multiprogramming, or multitasking**



(b) Multiprogramming with two programs



(c) Multiprogramming with three programs

- ☐ The most notable feature that is useful for multiprogramming is the hardware that supports I/O interrupts and DMA (direct memory access)
- ☐ With interrupt-driven I/O or DMA, the processor can issue an I/O command for one job and proceed with the execution of another job while the I/O is carried out by the device controller
- ☐ In addition, if several jobs are ready to run, the processor must decide which one to run, this decision requires an algorithm for scheduling

4. Time-Sharing Systems[Third Generation]

- ☐ In time sharing systems the processor time is shared among multiple users
- ☐ multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation
- ☐ If there are n users actively requesting service at one time, each user will only see on the average 1/n of the effective computer capacity

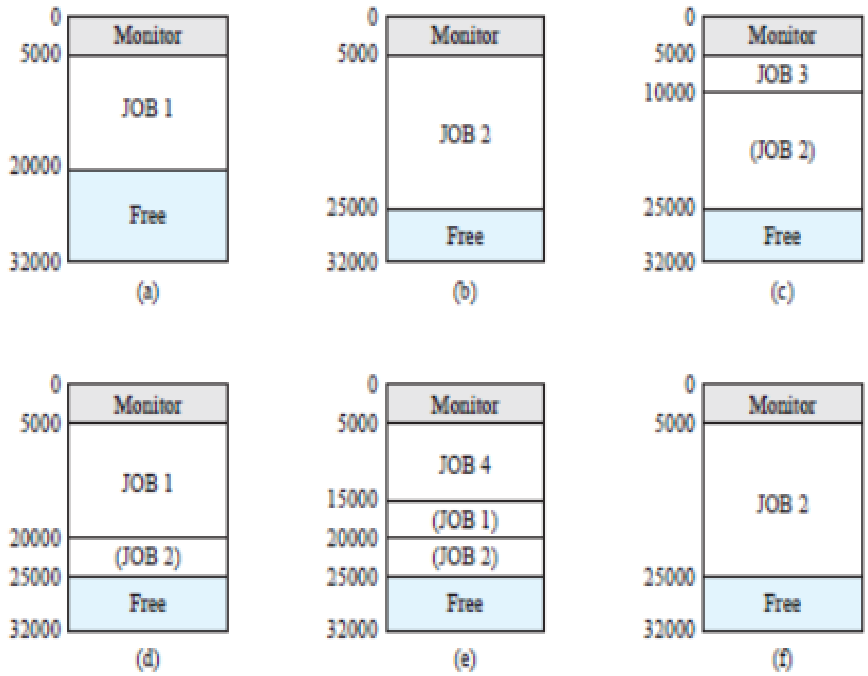
Batch Multiprogramming Vs Time Sharing systems

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

- One of the first time-sharing operating systems to be developed was the Compatible Time-Sharing System (CTSS)
- The system ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that. When control was to be assigned to an interactive user, the user's program and data were loaded into the remaining 27,000 words of main memory
- A program was always loaded to start at the location of the 5000th word
- A system clock generated interrupts at a rate of approximately one every 0.2 seconds
- At each clock interrupt, the OS regained control and could assign the processor to another user. This technique is known as **time slicing**

Example: Assume that there are four interactive users with the following memory requirements, in words:

JOB1: 15,000
JOB2: 20,000
JOB3: 5000
JOB4: 10,000



☐

i) Initially, the monitor loads JOB1 and transfers control to it.
ii) Later, the monitor decides to transfer control to JOB2. Because JOB2 requires more memory than JOB1, JOB1 must be written out first, and then JOB2 can be loaded.
iii) Next, JOB3 is loaded in to be run. However, because JOB3 is smaller than JOB2, a portion of
JOB2 can remain in memory, reducing disk write time.
iv) Later, the monitor decides to transfer control back to JOB1.An additional portion of JOB2 must be written out when JOB1 is loaded back into memory.
v) When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained.
vi) At this point, if either JOB1 or JOB2 is activated, only a partial load will be required. In this example, it
is JOB2 that runs next. This requires that JOB4 and the remaining resident portion of JOB1 be written out and that the missing portion of JOB2 be read in.

☐

5. MULTIPROCESSOR AND MULTICORE ORGANIZATION[Fourth Generation]

- ☐ A processor executes programs by executing machine instructions in sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results)

- In order to achieve performance and reliability, the concept of parallelism has been introduced in the computers which include symmetric multiprocessors, multicore computers and clusters.
- The multiple-processor systems in use today are of two types.
  - **Asymmetric multiprocessing**, in which each processor is assigned a specific task.A boss processor, controls the system; the other processors either look to the boss for instruction or have predefined tasks. **This scheme defines a boss–worker relationship.** The boss processor schedules and allocates work to the worker processors.
  - **Symmetric multiprocessing** (SMP), in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors.

Symmetric Multiprocessors:

- An SMP can be defined as a stand-alone computer system with the following characteristics:
  1. There are two or more similar processors of comparable capability.
  2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.
  3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
  4. All processors can perform the same functions
  5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.
- **Advantages of Symmetric multiprocessors:**

  1. Increased throughput.

  By increasing the number of processors, we expect to get more work done in less time. If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type..
- **2. Economy of scale**. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.
- **3. Increased reliability**. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system runs slower,

rather than failing altogether. Increased reliability of a computer system is crucial in many applications.

- ☐ The ability to continue providing service proportional to the **level of surviving hardware** is called **Graceful Degradation**. Some systems go beyond graceful degradation and are called **fault tolerant**, because they can suffer a failure of any single component and still continue operation.

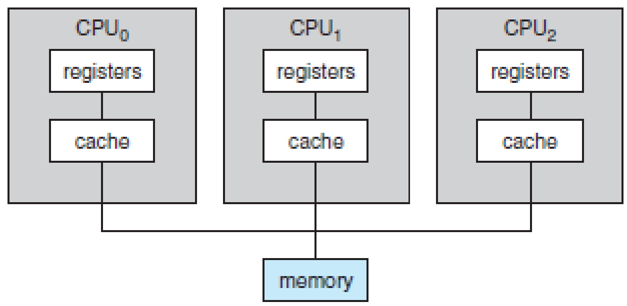


Figure 1.6 Symmetric multiprocessing architecture.

- ☐ **The Disadvantage of symmetric multiprocessor includes**

1) If one processor fails then it will affect in the speed
2) multiprocessor systems are expensive
3) Complex OS is required
4) Large main memory required.

MULTICORE ORGANIZATION

- ☐ A dual-core design contains two cores on the same chip.
- ☐ In this design, each core has its own register set as well as its own local cache. Other designs might use a shared cache or a combination of local and shared caches.
- ☐ Performance has also been improved by the increased complexity of processor design to exploit parallelism in instruction execution and memory access.

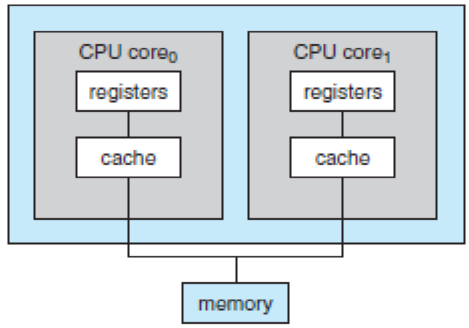


Figure 1.7 A dual-core design with two cores placed on the same chip.

- An example of a multicore system is the Intel Core i7, which includes four x86 processors, each with a dedicated L2 cache, and with a shared L3 cache

Difference between Multiprocessor and Multicore

- The main difference between multiprocessor and multicore systems lies in their hardware structure.
- Multicore systems have multiple processing units (cores) integrated onto a single integrated circuit (chip),
- while multiprocessor systems have multiple separate physical processors, each with its own core, connected to the same motherboard.

6. Distributed Systems

- A distributed system is a collection of physically separate, possibly heterogeneous computer systems that are networked to provide users with access to the various resources that the system maintains
- Access to a shared resource increases computation speed, functionality, data availability, and reliability
- Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver
- Others make users specifically invoke network functions
- Generally, systems contain a mix of the two modes—for example FTP and NFS
- A network, in the simplest terms,is a communication path between two or more systems.
- Distributed systems depend on networking for their functionality
- Independent Nodes - Each computer in the distributed system has its own operating system and local resources
- Communication Network These nodes are connected through a network (LAN, WAN, etc.), enabling them to communicate and exchange information

- ☐ Distributed OS Software:
  The distributed operating system software manages the interaction and coordination between these nodes, presenting a unified view to the user.
- ☐ Resource Sharing:
  It allows users and applications to access resources (like CPU, memory, storage, and peripherals) on any node in the network as if they were local.
- ☐ Transparency:
- ☐ DOS aims to hide the underlying distribution of resources, making it appear as if the system is a single, centralized machine
- ☐ Scalability:
  DOS systems are designed to be scalable, meaning they can easily accommodate the addition of new nodes without significant disruption.
- ☐ Fault Tolerance:
  By distributing tasks and resources across multiple nodes, DOS can continue functioning even if some nodes fail.
- ☐

7. Network Operating Systems

OPERATING SYSTEM STRUCTURE

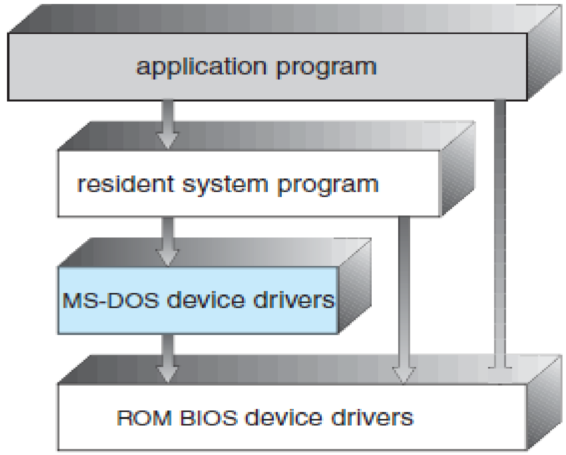
- ☐ The operating systems are large and complex. A common approach is to partition the task into small components, or modules, rather than have one **monolithic** system.
- ☐ The structure of an operating system can be defined as the following structures.
  - ☐ Simple structure
  - ☐ Layered approach
  - ☐ Microkernels
  - ☐ Modules
  - ☐ Hybrid systems

Simple structure:

- ☐ The Simple structured operating systems do not have a well defined structure. These systems will be simple, small and limited systems.

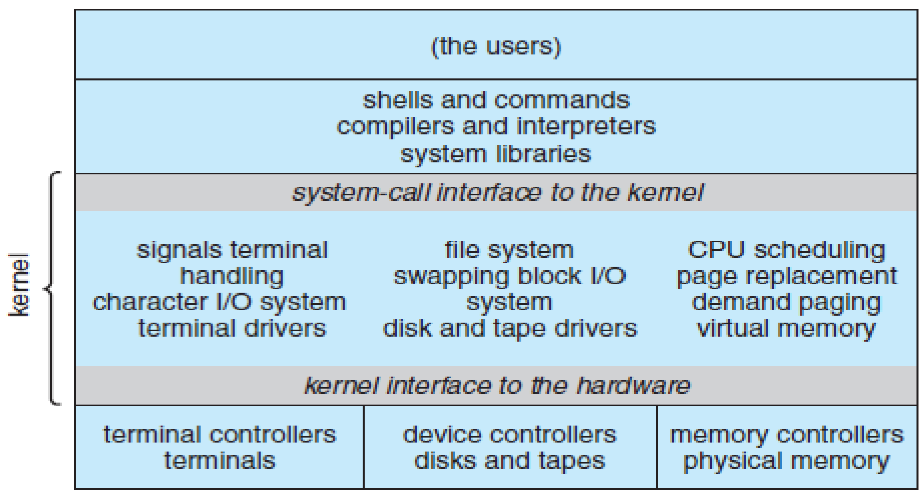
Example: MS-DOS.

- ☐ In MS-DOS, the interfaces and levels of functionality are not well separated.
- ☐ In MS-DOS application programs are able to access the basic I/O routines. This causes the entire system to crash when user programs fail.



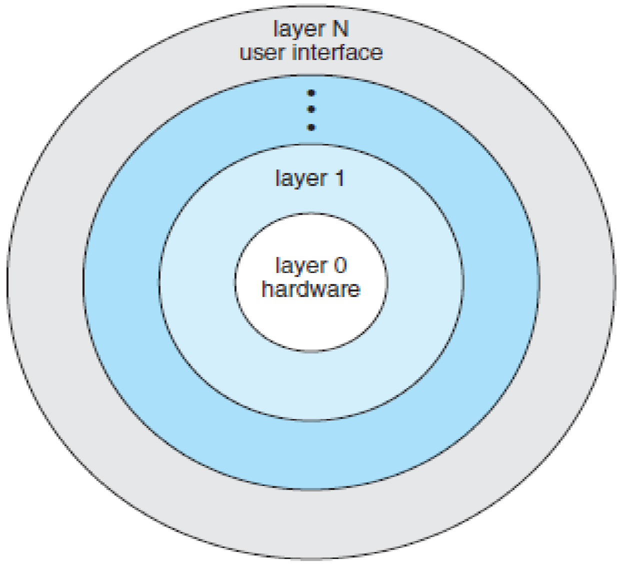
Example: Traditional UNIX OS

- ☐ It consists of two separable parts: the kernel and the system programs.
- ☐ The kernel is further separated into a series of interfaces and device drivers
- ☐ The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.
- ☐ This **monolithic structure** was difficult to implement and maintain.



Layered approach:

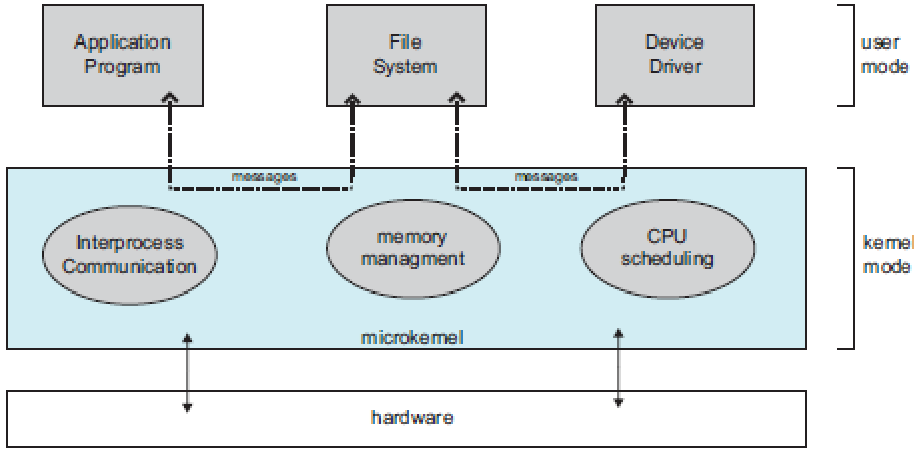
- ☐ A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.



- An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data.
- The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers.
- Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do.
- The major difficulty with the layered approach involves appropriately defining the various layers because a layer can use only lower-level layers.
- A problem with layered implementations is that they tend to be less efficient than other types.

Microkernels:

- In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach.
- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs.

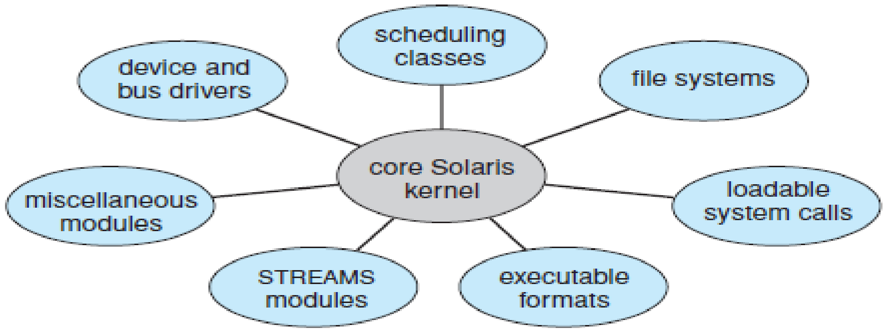


- Microkernel provide minimal process and memory management, in addition to a communication facility.
- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.
- The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.
- One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.
- The performance of microkernel can suffer due to increased system-function overhead.

Modular Structure

- The best current methodology for operating-system design involves using **loadable kernel modules**
- The kernel has a set of core components and links in additional services via modules, either at boot time or during run time.
- The kernel provides core services while other services are implemented dynamically, as the kernel is running.
- Linking services dynamically is more comfortable than adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.

Example: Solaris OS



- ☐ The Solaris operating system structure is organized around a core kernel with seven types of loadable kernel modules:
    - ☐ Scheduling classes
    - ☐ File systems
    - ☐ Loadable system calls
    - ☐ Executable formats
    - ☐ STREAMS modules
    - ☐ Miscellaneous
    - ☐ Device and bus drivers

Hybrid Systems:

- ☐ The Operating System combines different structures, resulting in hybrid systems that address performance, security, and usability issues.
- ☐ They are monolithic, because having the operating system in a single address space provides very efficient performance.
- ☐ However, they are also modular, so that new functionality can be dynamically added to the kernel.
- ☐ Example: Linux and Solaris are monolithic (simple) and also modular, IOS.
- ☐ Apple IOS Structure

Structure Type	Description	Examples
Monolithic Kernel	All core services run in kernel space for speed	Linux, older UNIX
Microkernel	Minimal kernel; services run in user space	QNX, seL4
Hybrid Kernel	Mix of monolithic and microkernel features	Windows NT, macOS
Modular Structure	Kernel with loadable modules for flexibility	Linux, Solaris
Layered Structure	OS divided into layers with clear interfaces	UNIX (partially)
Virtual Machines	OS runs on virtualized hardware	VMware, Hyper-V

SYSTEM CALLS

- The system call provides an interface to the operating system services.
- Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls.
- Systems execute thousands of system calls per second. Application developers design programs according to an **application programming interface (API)**.
- For most programming languages, the Application Program Interface provides a **system call interface** that serves as the link to system calls made available by the operating system.
- The system-call interface intercepts function calls in the API and invokes the necessary system calls within the Operating system.
- **Example: System calls for writing a simple program to read data from one file and copy them to another file**

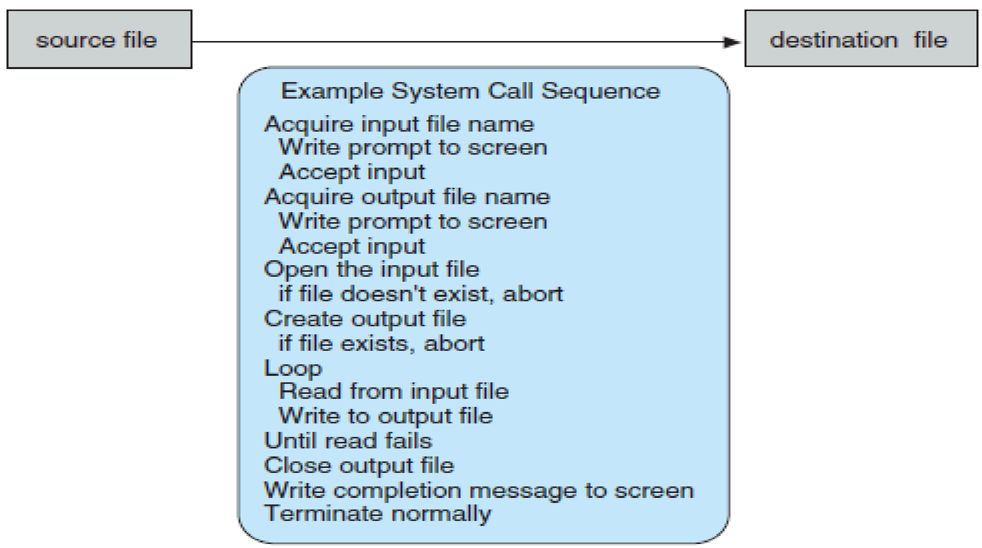


Figure 2.5 Example of how system calls are used.

- The caller of the system call knows nothing about how the system call is implemented or what it does during execution.
- The caller need only obey the API and understand what the operating system will do as a result of the execution of that system call.

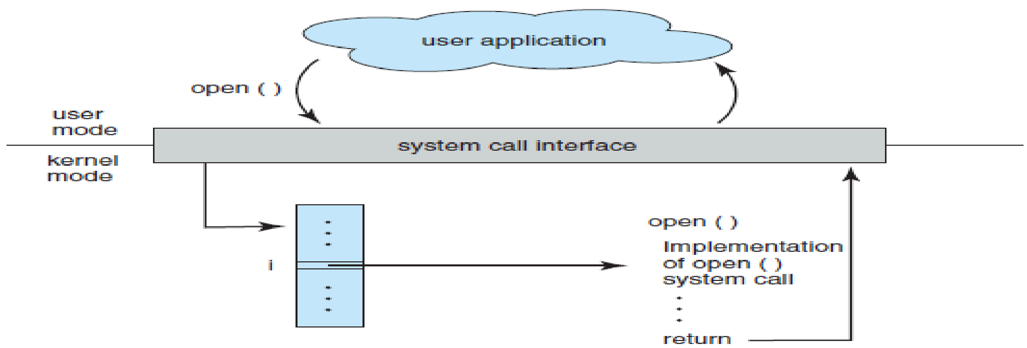


Figure 2.6 The handling of a user application invoking the open() system call.

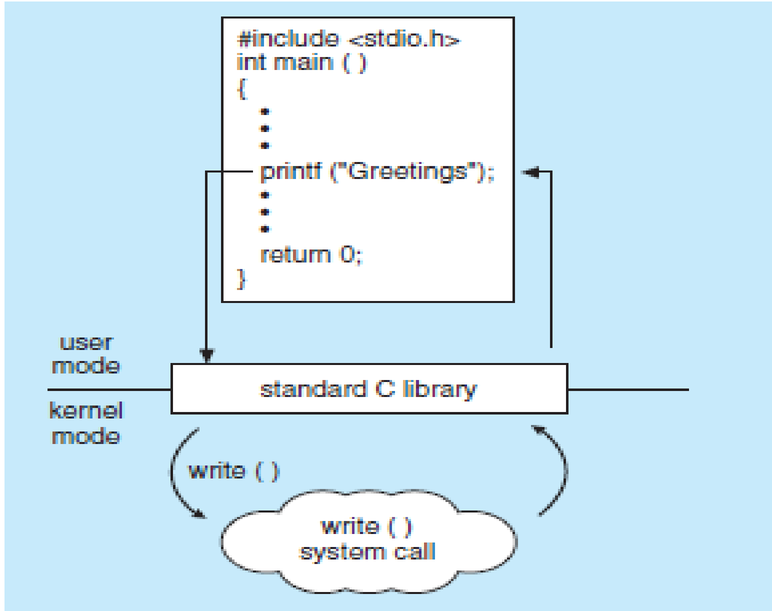
- Three general methods are used to pass parameters to the operating system
  - pass the parameters in registers
  - parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register
  - Parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system.

Types of System Calls:

- System calls can be grouped roughly into six major categories
  - **Process control**,
  - **File manipulation,**
  - **Device manipulation**,
  - **Information maintenance**,
  - **Communications**,
  - **Protection**.

PROCESS CONTROL

- A Running program needs to be able to halt its execution either normally (**end ()**) or abnormally (**abort()**).
- Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter.
- A process or job executing one program may want to **load()** and **execute()** another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command.



- If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution that requires to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (**get process attributes()** and **set process attributes()**).
- We may also want to terminate a job or process that we created (**terminate process()**) if we find that it is incorrect or is no longer needed.
- The System -calls associated with process control includes
  - **end, abort**
  - **load, execute**
  - **create process, terminate process**
  - **get process attributes, set process attributes**
  - **Wait for time**
  - **wait event, signal event**
  - **allocate and free memory**

- When a process has been created, we may want to wait for a certain amount of time to pass (**wait time()**) or we will want to wait for a specific event to occur (**wait event()**).
- The jobs or processes should then signal when that event has occurred (**signal event()**
- To start a new process, the shell executes a **fork()** system call. Then, the selected program is loaded into memory via an **exec()** system call, and the program is executed
- When the process is done, it executes an **exit()** system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code.

FILE MANAGEMENT

- In order to work with files, we first need to be able to **create ()** and **delete ()** files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to **open()** it and to use it.
- We may also **read ()**, **write ()**, or **reposition ()**.Finally, we need to **close ()** the file, indicating that we are no longer using it.

- ☐ In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary.
- ☐ File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, **getfileattributes ()** and **setfileattributes ()**, are required for this function.
- ☐ The System calls associated with File management includes
  - ☐ **File management**
  - ☐ **create file, delete file**
  - ☐ **open, close**
  - ☐ **read, write, reposition**
  - ☐ **get file attributes, set file attributes**

DEVICE MANAGEMENT:

- ☐ A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.
- ☐ A system with multiple users may require us to first **request()** a device, to ensure exclusive use of it.
- ☐ After we are finished with the device, we **release()** it. These functions are similar to the open() and close() system calls for files.

Once the device has been requested (and allocated to us), we can **read()**, **write()**, and (possibly) **reposition()** the device, just as we can with files.

- ☐ I/O devices are identified by special file names, directory placement, or file attributes.
- ☐ The System calls associated with Device management includes
  - **request device, release device**
  - **read, write, reposition**
  - **get device attributes, set device attributes**
  - **logically attach or detach devices**

INFORMATION MAINTENANCE:

- ☐ Many system calls exist simply for the purpose of transferring information between the user program and the operating system.
- ☐ Example, most systems have a system call to return the **current time()** and **date()**.
- ☐ Other system calls may return information about the system, such as the **number of current users, the version number of the operating system, the amount of free memory or disk space**, and so on.
- ☐ Many systems provide system calls to **dump()** memory. This provision is useful for debugging.
- ☐ Many operating systems provide a **time profile of a program** to indicate the amount of time that the program executes at a particular location or set of locations.
- ☐ The operating system **keeps information about all its processes, and system calls** are used to access this information.
- ☐ Generally, calls are also used to reset the process information (get process attributes() and set process attributes()).
- ☐ The System calls associated with Device management includes

- **get time or date, set time or date**
- **get system data, set system data**
- **get process, file, or device attributes**
- **set process, file, or device attributes**

COMMUNICATION:

- There are two common models of Interprocess communication: the message passing model and the shared-memory model.
- In the **message-passing model**, the communicating processes exchange messages with one another to transfer information.
- Messages can be exchanged between the processes either directly or indirectly through a common mailbox.
- Each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process. The **get hostid()** and **get processid()** system calls do this translation.
- The recipient process usually must give its permission for communication to take place with an **accept connection ()** call.
- The source of the communication, known as the **client**, and the receiving daemon, known as a **server,** then exchange messages by using **read message()** and **write message()** system calls.
- The **close connection()** call terminates the communication
- In the **shared-memory model**, processes use shared memory **create()** and shared memory **attach()** system calls to create and gain access to regions of memory owned by other processes.

- The system calls associated with communication includes,
    - **create, delete communication connection**
    - **send, receive messages**
    - **Transfer status information**
    - **attach or detach remote devices**

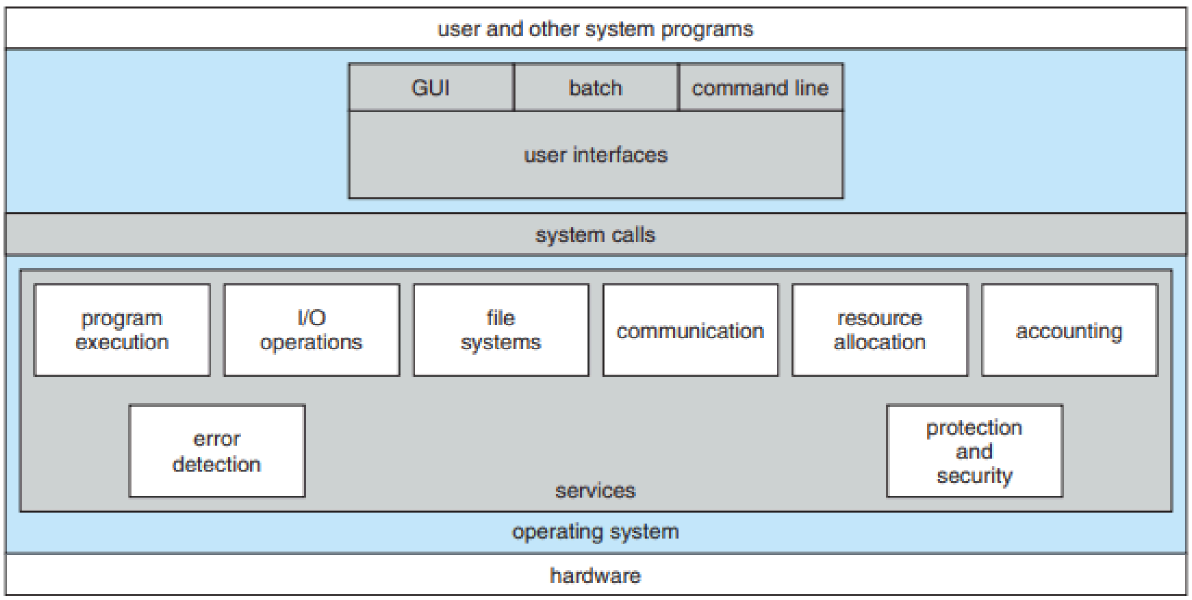
PROTECTION:

- Protection provides a mechanism for controlling access to the resources provided by a computer system.
- System calls providing protection include **set permission ()** and **get permission ()**, which manipulate the permission settings of resources such as files and disks.
- The **allow user ()** and **deny user ()** system calls specify whether particular users can—or cannot—be allowed access to certain resources.

Operating-System Services

- An OS provides certain services to programs and to the users of those programs
-
- **1. Services helpful to the user**
- **User interface**

- Almost all OS have interface
- **command-line interface (CLI)** - uses text commands and a method for entering them
- **batch interface** - commands and directives to control those commands are entered into files, and those files are executed
- **graphical user interface (GUI)** - the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text
- **Program Execution**
- The system must be able to load a program into memory and to run that program
- The program must be able to end its execution, either normally or abnormally



- **Figure 2.1** A view of operating system services.
- **I/O operations**
- A running program may require I/O, which may involve a file or an I/O device
- special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen)
- users usually cannot control I/O devices directly
- the operating system must provide a means to do I/O
- **File-system manipulation**
- programs need to read and write files and directories
- need to create and delete them by name, search for a given file, and list file information

- [ ] permissions management to allow or deny access to files or directories based on file ownership
- [ ] provide a variety of file systems, for specific features or performance characteristics
- [ ] **Communications**
- [ ] one process needs to exchange information with another process
- [ ] may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network
- [ ] implemented via
    - [ ] shared memory - two or more processes read and write to a shared section of memory
    - [ ] message passing - packets of information in predefined formats are moved between processes by the operating system
- [ ] **Error detection**
- [ ] The operating system needs to be detecting and correcting errors constantly
- [ ] Errors may occur in the
    - [ ] CPU
    - [ ] Memory hardware (mem. Error or power failure)
    - [ ] I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer)
    - [ ] user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time)
- [ ] For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing
    - [ ] Sometimes, it has no choice but to halt the system
    - [ ] At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct
- [ ] **2. Services for ensuring the efficient operation of the system**
- [ ] **Resource allocation**
- [ ] When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them
- [ ] types of resources
    - [ ] CPU cycles
    - [ ] main memory
    - [ ] file storage
    - [ ] I/O devices
- [ ] **Accounting**

- record keeping
  - which users use how much and what kinds of computer resources
- for accumulating usage statistics
- to reconfigure the system
- to improve computing services
- **Protection and security**
- The owners of information stored in a multiuser or networked computer system may want to control use of that information
- it should not be possible for one process to interfere with the others or with the operating system itself
- ensuring that all access to system resources is controlled
- requiring each user to authenticate, through password
-

User and Operating-System Interface

- Two fundamental approaches
- **1. Command Interpreters**
- Some operating systems include the command interpreter in the kernel
- Others, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on
- On systems with multiple command interpreters to choose from, the interpreters are known as shells
- on UNIX and Linux systems, a user may choose among several different shells, including the Bourne shell, C shell, Bourne-Again shell, Korn shell, and others

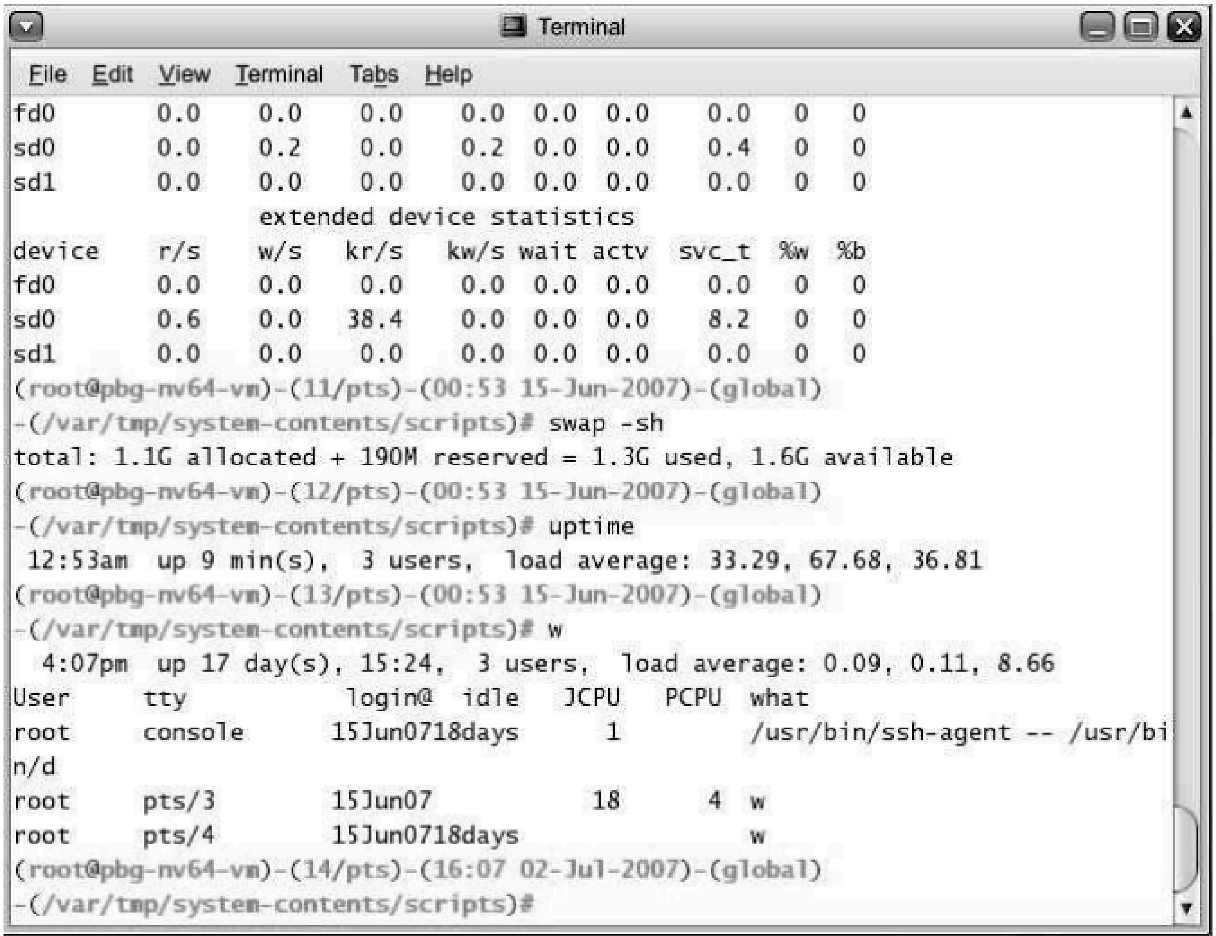


Figure 2.2 The Bourne shell command interpreter in Solrais 10.

- The main function of the command interpreter is to get and execute the next user-specified command.
- Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on
- These commands can be implemented in two general ways
- 1. — the command interpreter itself contains the code to execute the command
- For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call

- ☐ 2. — implements most commands through system programs. In this case, the command interpreter does not understand the command in any way
- ☐ the UNIX command to delete a file **rm file.txt** would search for a file called rm, load the file into memory, and execute it with the parameter file.txt
- ☐ In this way, programmers can add new commands to the system easily by creating new files
- ☐ **2. Graphical User Interfaces**
- ☐ users employ a mouse-based window-and-menu system characterized by a **desktop** metaphor
- ☐ The user moves the mouse to position its pointer on images, or **icons**, (represent programs, files, directories, and system functions) on the screen
- ☐ clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**
- ☐ Because a mouse is impractical for most mobile systems, smartphones and handheld tablet computers typically use a **touchscreen interface**
- ☐ users interact by making gestures on the **touchscreen**—for example, pressing and swiping fingers across the screen

System Programs

- ☐ System programs, also known as system utilities, provide a convenient environment for program development and execution
- ☐ Some of them are simply user interfaces to system calls
- ☐ **Categories**
- ☐ **File management**
- ☐ These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- ☐ **Status information**
- ☐ Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information

- ☐ Others are more complex, providing detailed performance, logging, and debugging information
- ☐ Some systems also support a registry, which is used to store and retrieve configuration information
- ☐ **File modification**
- ☐ Several text editors may be available to create and modify the content of files stored on disk or other storage devices
- ☐ There may also be special commands to search contents of files or perform transformations of the text
- ☐ **Programming-language support**
- ☐ Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download
- ☐ **Program loading and execution**
- ☐ Once a program is assembled or compiled, it must be loaded into memory to be executed.
- ☐ The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders.
- ☐ Debugging systems for either higher-level languages or machine language are needed as well
- ☐ **Communications**
- ☐ provide the mechanism for creating virtual connections among processes, users, and computer systems
- ☐ allow users to send messages to one another’s screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another
- ☐ **Background services**
- ☐ All general-purpose systems have methods for launching certain system-program processes at boot time
- ☐ Some of these processes terminate after completing their tasks, while others continue to run until the system is halted
- ☐ Constantly running system-program processes are known as services, subsystems, or daemons

Operating-System Design and Implementation

- ☐ We discuss problems we face in designing and implementing an operating system
- ☐ **Design Goals**
- ☐ At the highest level, the design of the system will be affected by the
    - ☐ choice of hardware and the
    - ☐ type of system: batch, time sharing, single user, multiuser, distributed, real time, or general purpose
- ☐ The requirements can, however, be divided into two basic groups
- ☐ **user goals**
    - ☐ The system should be convenient to use, easy to learn and to use, reliable, safe, and fast
- ☐ **system goals**
- ☐ easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient
- ☐ There is not a unique solution in the creation and design of operating systems, but the principles of **software engineering** can be taken as guidance
- ☐ **Principles**
- ☐ **1.Mechanisms and Policies**
- ☐ One important principle is the separation of policy from mechanism
- ☐ Mechanisms determine **how to do** something(eg. Timer construct to provide protection)
- ☐ Policies determine **what will be done**(eg.How long the timer is set for a user)
- ☐ Policies are likely to change across places or over time
- ☐ each change in policy would require a change in the underlying mechanism
- ☐ A general mechanism insensitive to changes in policy would be more desirable
- ☐ A change in policy would then require redefinition of only certain parameters of the system
- ☐ Eg. Mechanism - Giving priority to certain programs, Policy - Can support either Priority to CPU-intensive or I/O intensive programs

- ☐ Policy decisions are important for all resource allocation
- ☐ Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made
- ☐ **2.Implementation**
- ☐ operating systems are collections of many programs, written by many people over a long period of time
- ☐ Early operating systems were written in assembly language.
- ☐ Now, although some operating systems are still written in assembly language, most are written in a higher-level language such as C or an even higher-level language such as C++
- ☐ an operating system can be **written in more than one language**. The lowest levels of the kernel might be assembly language. Higher-level routines might be in C, and system programs might be in C or C++, in interpreted scripting languages like PERL or Python, or in shell scripts
- ☐ Advantages of using a higher-level language for implementing operating systems
    - ☐ the code can be written faster
    - ☐ is more compact
    - ☐ easier to understand and debug
    - ☐ improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation
    - ☐ operating system is far easier to port—to move to some other hardware
- ☐ Disadvantages of implementing an operating system in a higher-level language
    - ☐ reduced speed
    - ☐ increased storage requirements
- ☐ no longer a major issue in today's systems
    - ☐ a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code
    - ☐ Modern processors have deep pipelining and multiple functional units

- major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code
- Only a small amount of the code is critical to high performance;
    - the interrupt handler
    - I/O manager
    - memory manager
    - CPU scheduler
- bottleneck routines can be identified and can be replaced with assembly-language equivalents

Unit - 2

Process Management

- A process can be thought of as a program in execution
- A process will need certain resources—such as CPU time, memory, files, and I/O devices
- **operating-system processes** execute system code, and **user processes** execute user code
- The terms job and process are used almost interchangeably
- A process is more than the program code, which is sometimes known as the **text section**
- It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers
- A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables)
- a data section, which contains global variables
- A process may also include a heap, which is memory that is dynamically allocated during process run time

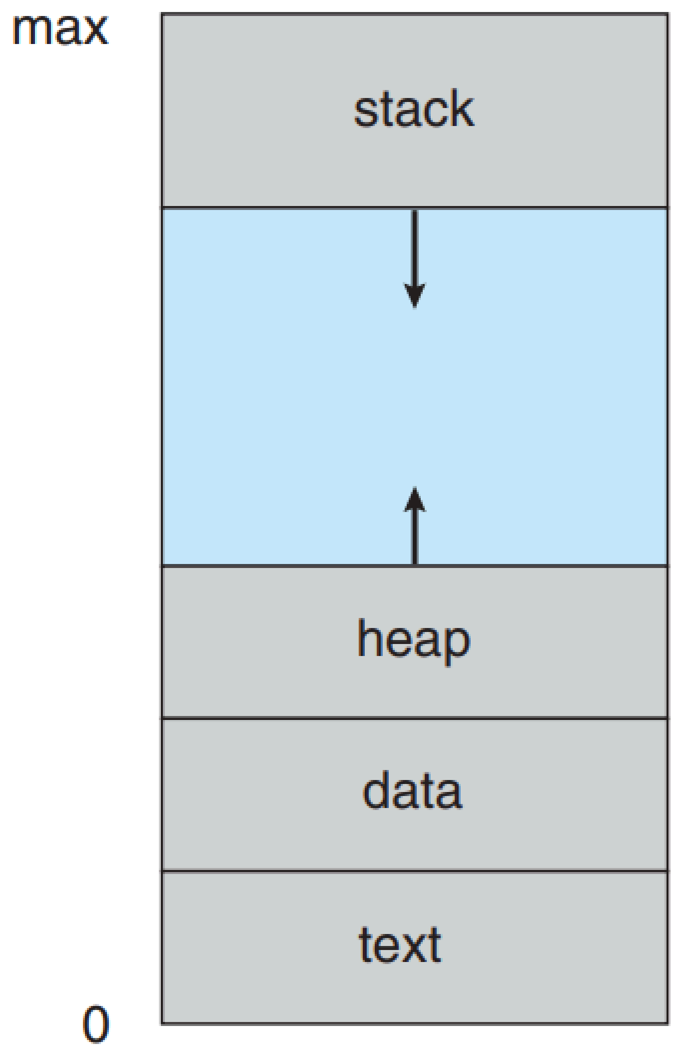


Figure 3.1 Process in memory.

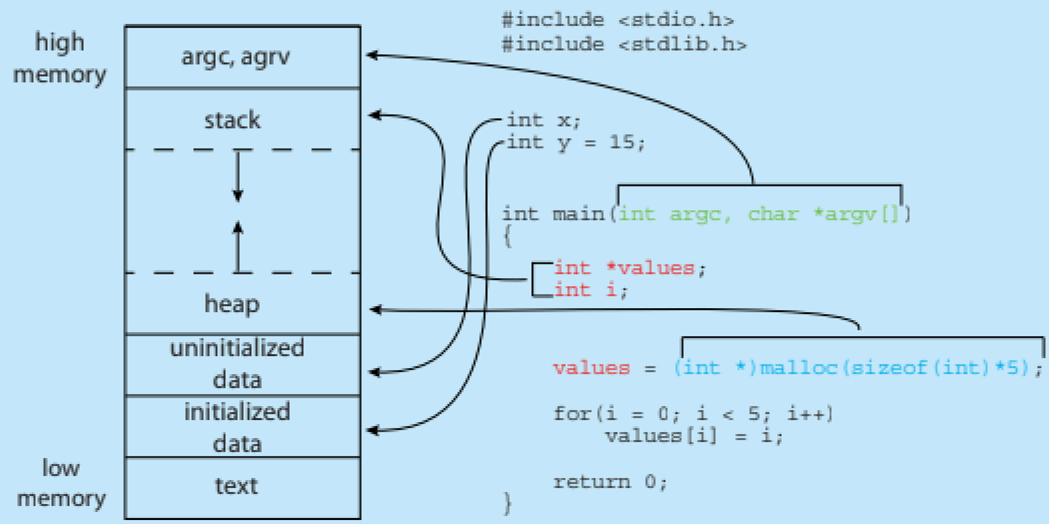
- ☐
- ☐ A program is a passive entity, such as a file containing a list of instructions stored on disk(often called an executable file)
- ☐ a process is an active entity
- ☐ Two processes may be associated with the same program
- ☐ They are considered two separate execution sequences.
- ☐ The text sections are equivalent
- ☐ the data, heap,and stack sections vary.
- ☐ a process itself can be an execution environment for othercode.

- Eg. Java programming environment JVM
- to run the compiled Java program Program.class
- We enter java Program
- runs the JVM as an ordinary process, which in turn executes the Java program Program in the virtual machine.

MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

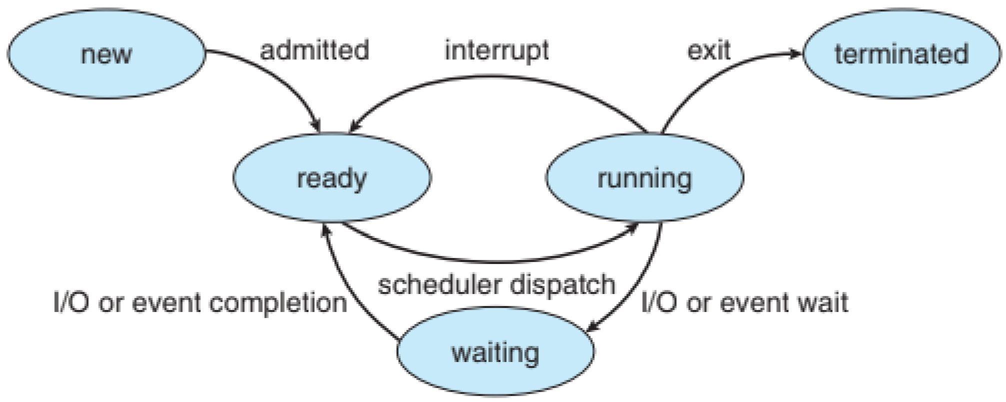
- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



Process State

- The state of a process is defined in part by the current activity of that process.
- A process may be in one of the following states:
- **New.** The process is being created.
- **Running.** Instructions are being executed.

- **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready.** The process is waiting to be assigned to a processor.
- **Terminated.** The process has finished execution.



- Figure 3.2 Diagram of process state.

Process Control Block

- Each process is represented in the operating system by a process control block (PCB)—also called a task control block
- It contains many pieces of information
- **Process state**. The state may be new, ready, running, waiting, halted, and so on
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables

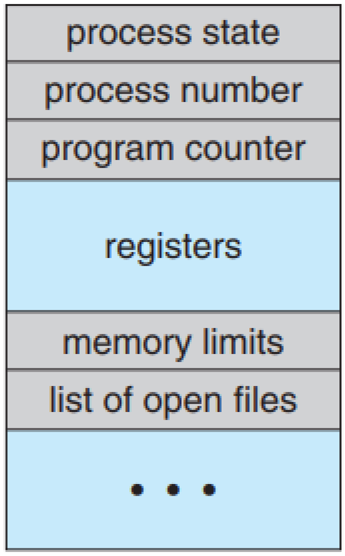


Figure 3.3 Process control block (PCB).

-

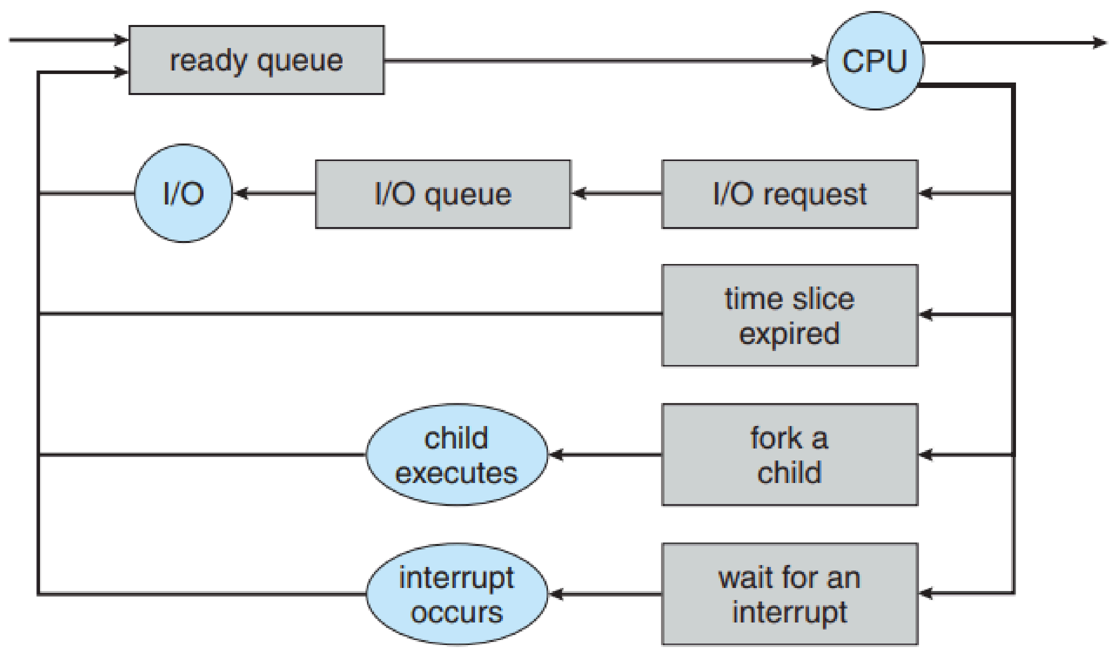
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on

Threads

- Process is a program that performs a **single thread** of execution
- For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker within the same process
- 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.
- On a system that supports threads, the PCB is expanded to include information for each thread

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program
- The process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU
- **Scheduling Queues**
- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system
- The processes that are **residing in main memory** and are ready and waiting to execute are kept on a list called the **ready queue -** stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list
- The list of processes waiting for a particular I/O device is called a **device queue** - each device has its own device queue



- **Figure 3.6** Queueing-diagram representation of process scheduling.
- A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**
- The process could issue an I/O request and then be placed in an **I/O queue**

- ☐ The process could create a new child process and wait for the child's termination
- ☐ The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue

Schedulers

- ☐ A process migrates among the various scheduling queues throughout its lifetime
- ☐ The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler
- ☐ The **long-term scheduler, or job scheduler**, selects processes from this pool and loads them into memory for execution
    - ☐ The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next - controls the degree of multiprogramming
    - ☐ An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations
    - ☐ A CPU-bound process, in contrast, generates I/O requests infrequently
    - ☐ The long-term scheduler select a good process mix of I/O-bound and CPU-bound processes
- ☐ The **short-term scheduler, or CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them
    - ☐ The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.
- ☐ Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling - **medium-term scheduler**
- ☐ The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) - reduce the degree of multiprogramming

- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**

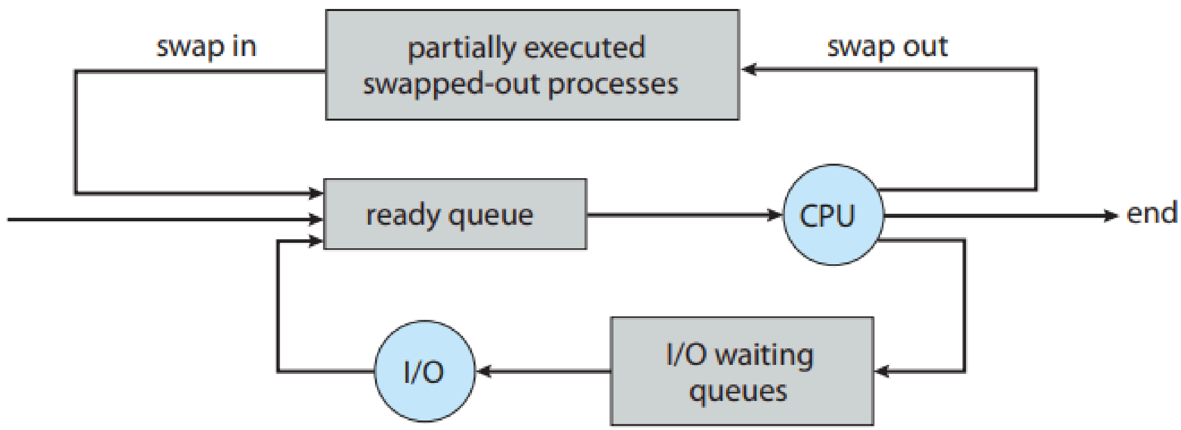


Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

- **Context Switch**
- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done
- The context is represented in the PCB of the process
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run

Operations on Processes

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically
- Process Creation
- a process may create several new processes
- the creating process is called a parent process
- The new processes are called the children of that process
- Each of these new processes may in turn create other processes, forming a tree of processes.

- identify processes according to a unique process identifier (or pid)

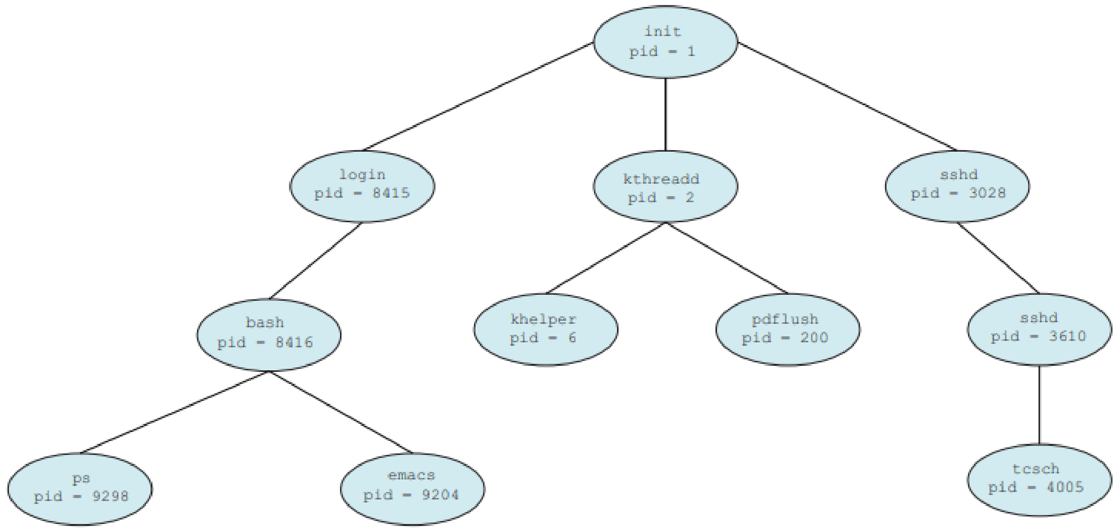


Figure 3.8 A tree of processes on a typical Linux system.

-
- The init process (which always has a pid of 1) serves as the root parent process for all user processes.
- Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like.
- two children of init—kthreadd and sshd. The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, khelper and pdflush)
- The sshd process is responsible for managing clients that connect to the system by using ssh
- The login process is responsible for managing clients that directly log onto the system
- when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process

- the parent process may pass along initialization data (input) to the child process
- When a process creates a new process, two possibilities for execution exist:
  - 1. The parent continues to execute concurrently with its children have terminated
  - 2. The parent waits until some or all of its children have terminated
- There are also two address-space possibilities for the new process:
  - 1. The child process is a duplicate of the parent process (it has the same program and data as the parent
  - 2. The child process has a new program loaded into it
- A new process is created by the **fork()** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process
- The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent. After a fork() system call, one of the two two processes typically uses the exec() system call to replace the process's memory space with a new program. The exec() system call loads a binary file into memo

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

int main()
{
pid_t pid;

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

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

    return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

- ☐
- ☐ Process Termination
- ☐ A process terminates when it finishes executing its final statement
- ☐ asks the operating system to delete it by using the exit() system call.
- ☐ All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.
- ☐ A parent may terminate the execution of one of its children for a variety of reasons, such as these:
- ☐ The child has exceeded its usage of some of the resources that it has been allocated.
- ☐ The task assigned to the child is no longer required.
- ☐ The parent is exiting, and the operating system does not allow a child to continue if its parent terminates

- if a process terminates (either normally or abnormally), thenall its children must also be terminated. This phenomenon, referred to as cascading termination.
- A parent process may wait for the termination of a child process by using the wait() system call
- The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child
- pid_t pid;
- int status;
- pid = wait(&status);
- A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process.
- In this case, child processes are known as **orphans**
-

CPU scheduling

- Several processes are kept in memory at one time.
- By switching the CPU among processes, the operating system can make the computer more productive.
-
- CPU –I/O Burst Cycle

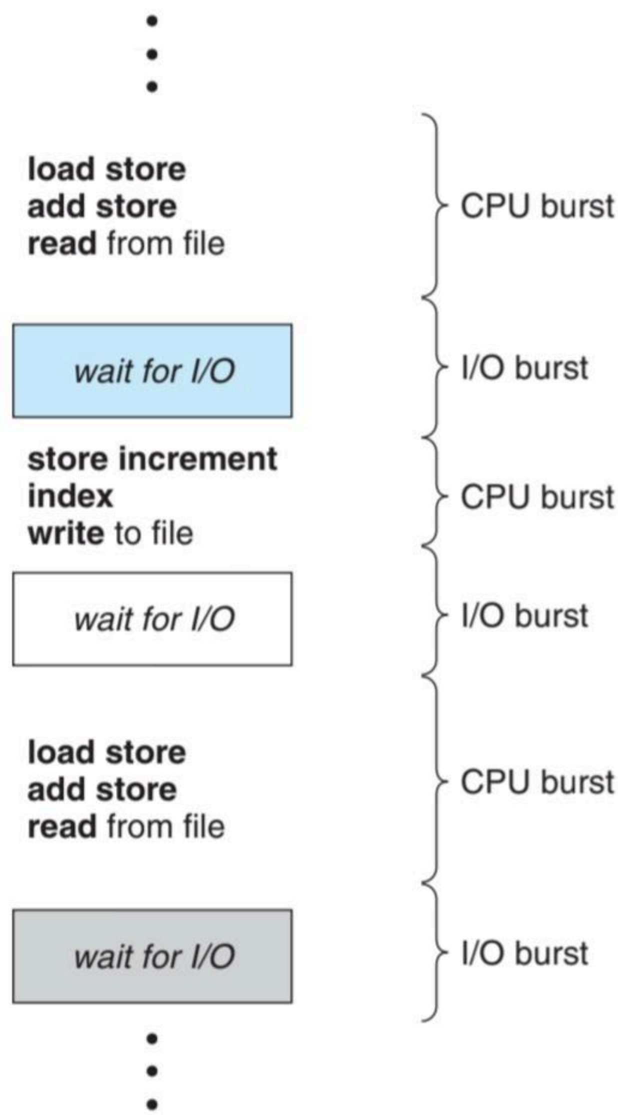


Figure 6.1 Alternating sequence of CPU and I/O bursts.

- ☐
- ☐
- ☐ Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler, or CPU scheduler.
- ☐ CPU-scheduling decisions may take place under the following four circumstances:

- 1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
- 2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
- 3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
- 4.When a process terminate
- When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative.
- Otherwise,it is preemptive
- Under **nonpreemptive scheduling**, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Virtually all modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.
- **Dispatcher**
- is the module that gives control of the CPU's core to the process selected by the CPU scheduler.
- involves the following:
    - Switching context from one process to another
    - Switching to user mode
    - Jumping to the proper location in the user program to resume that program
- **Scheduling Criteri**a
- 1. CPU utilization - We want to keep the CPU as busy as possible. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
- Throughput - the number of processes that are completed per time unit, called throughput.
- Turnaround time - The interval from the time of submission of a process to the time of completion is the turnaround time.
- Waiting time - Waiting time is the sum of the periods spent waiting in the ready queue.

- ☐ Response time - time from the submission of a request until the first response is produced.

Scheduling Algorithms

- ☐ First-Come First-Serve (FCFS) scheduling - Refer your Class notebook for the problems we have discussed
- ☐ Non-Preemptive Shortest Job First Scheduling (SJF) - Refer your Class notebook for the problems we have discussed
- ☐ Preemptive Shortest Job First Scheduling (SJF) or SRTF (Shortest Remaining Time First) - Refer your Class notebook for the problems we have discussed
- ☐ Round Robin (RR) scheduling - Refer your Class notebook for the problems we have discussed
- ☐ Non-Preemptive Priority Scheduling - Refer your Class notebook for the problems we have discussed
- ☐ Preemptive Priority Scheduling - Refer your Class notebook for the problems we have discussed

Threads

- ☐ A thread is a basic unit of CPU utilization
- ☐ it comprises a thread ID, a program counter (PC), a register set, and a stack.
- ☐ It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- ☐ A traditional process has a single thread of control.
- ☐ If a process has multiple threads of control, it can perform more than one task at a time.

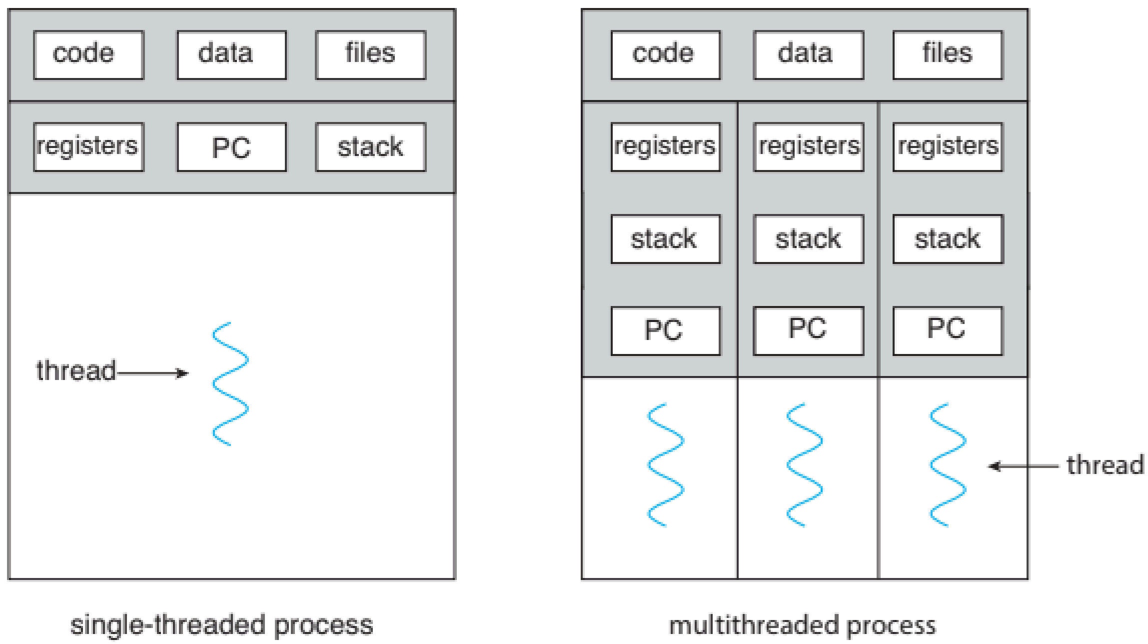


Figure 4.1 Single-threaded and multithreaded processes.

- **Multithreading Models**
- User threads are supported above the kernel and are managed without kernel support

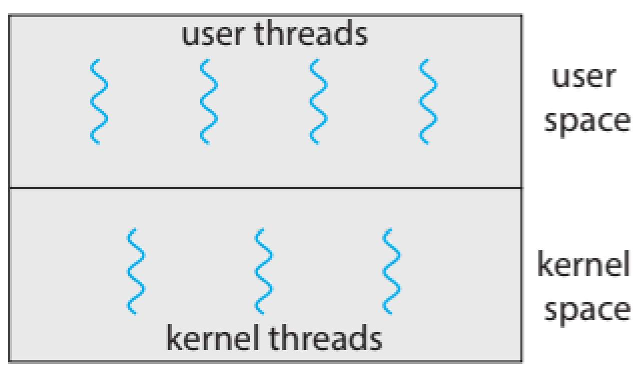


Figure 4.6 User and kernel threads.

- kernel threads are supported and managed directly by the operating system.
- 1. **Many-to-One Model**

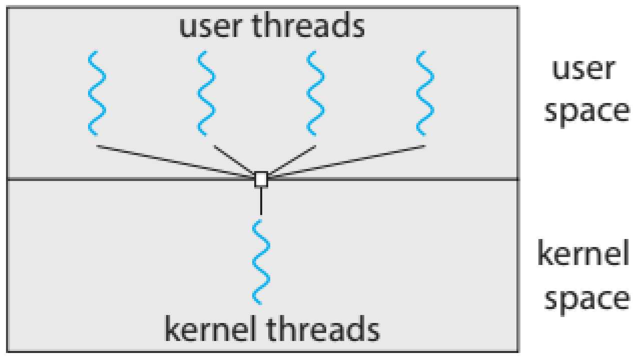


Figure 4.7 Many-to-one model.

- ☐
- ☐ maps many user-level threads to one kernel thread.
- ☐ Thread management is done by the thread library in user space
- ☐ the entire process will block if a thread makes a blocking system call.
- ☐ one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.
- ☐
- ☐ 2. **One-to-One Model**

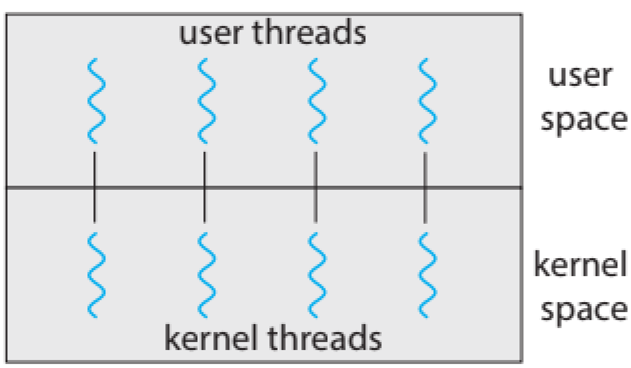


Figure 4.8 One-to-one model.

- ☐
- ☐ maps each user thread to a kernel thread.
- ☐ provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- ☐ allows multiple threads to run in parallel on multiprocessors.

- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system.
-
- **3. Many-to-Many Model**

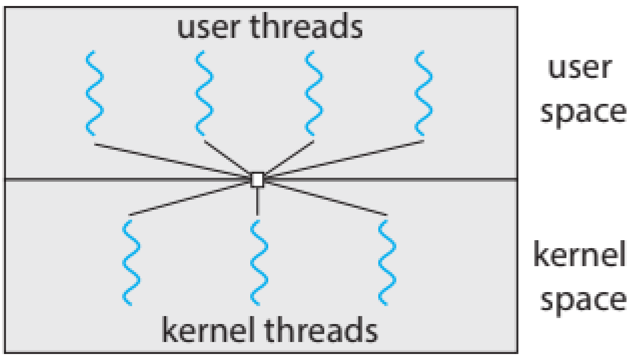


Figure 4.9 Many-to-many model.

-
- multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine
-
- **Two-level model**

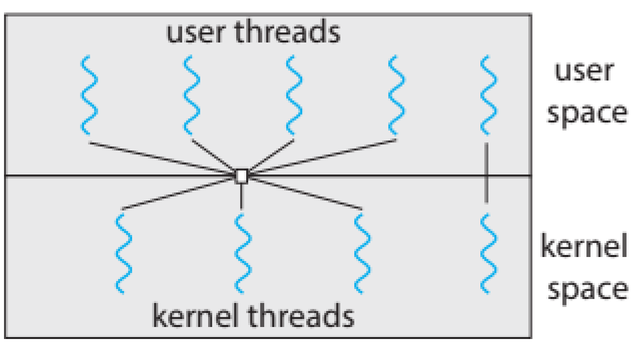


Figure 4.10 Two-level model.

-
- One variation on the many-to-many model
- still multiplexes many user-level threads to a smaller or equal number of kernel threads

- ☐ but also allows a user-level thread to be bound to a kernel thread.
- ☐

Process Synchronization

- ☐ A system typically consists of several threads running either concurrently or in parallel.
- ☐ Threads often share user data
- ☐ A race condition exists when access to shared data is not controlled, possibly resulting in corrupt data values.
- ☐ Process synchronization involves using tools that control access to shared data to avoid race conditions.
- ☐ A cooperating process
  - ☐ is one that can affect or be affected by other processes executing in the system
  - ☐ Cooperating processes can either directly share a logical address space (that is, both code and data) or
  - ☐ be allowed to share data only through shared memory or message passing

The Critical-Section Problem

- ☐ Consider a system consisting of n processes {P0, P1, ..., Pn−1}.
- ☐ Each process has a segment of code, called a critical section, in which the process may be accessing — and updating — data that is shared with at least one other process
- ☐ The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section
- ☐ Each process must request permission to enter its critical section. The section of code implementing this request is the entry section.
- ☐ The critical section may be followed by an exit section
- ☐ The remaining code is the remainder section.

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Figure 6.1 General structure of a typical process.

- ☐
- ☐ A solution to the critical-section problem must satisfy the following three requirements:
- ☐ **1. Mutual exclusion.** If process Pi 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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
- ☐ **3. Bounded waiting**. There exists a bound, or limit, 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.

Peterson's Solution

- ☐ Software based solution
- ☐ Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- ☐ Peterson's solution requires the two processes to share two data items: int turn; boolean flag[2];

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

- Figure 6.3 The structure of process P_i in Peterson's solution.

Mutex Locks

- software tool to solve the critical-section problem
- We use the mutex lock to protect critical sections and thus prevent race conditions
- a process must acquire the lock before entering a critical section(acquire()function)
- it releases the lock when it exits the critical section(release()function)
- A mutex lock has a boolean variable available whose value indicates if the lock is available or not
- If the lock is available, a call to acquire()succeeds, and the lock is then considered unavailable
- A process that attempts to acquire an unavailable lock is blocked until the lock is released.

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

- **Figure 6.10** Solution to the critical-section problem using mutex locks.

The definition of acquire() is as follows:

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
```

The definition of release() is as follows:

```
release() {
   available = true;
}
```

-
- Calls to either acquire() or release() must be performed atomically
- A lock is considered **contended** if a thread blocks while trying to acquire the lock
- If a lock is available when a thread attempts to acquire it, the lock is considered **uncontended**
- **Disadvantage**
    - **busy waiting** - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire()
- The type of mutex lock we have been describing is also called a spin lock because the process "spins" while waiting for the lock to become available

Semaphores

- A **semaphore S** is an integer variable that,
- It can be initialized
- accessed only through two standard atomic operations: **wait() and signal()**
- definition of wait()

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- definition of signal()

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

- when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.
- The Wait Operation is used for deciding the condition for the process to enter the critical state or wait for execution of process
- If the Semaphore value is equal to zero then the Process has to wait for the Process to exit the Critical Section Area
- if the Semaphore value is greater than zero or positive then the Process can enter the Critical Section Area
- **Types of Semaphores**
- **Counting Semaphores**
- These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented
- **Binary Semaphores**
- The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore

is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores

- **Advantages of Semaphores**
- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent
- **Example**, consider two concurrently running processes:
- we require that S2 be executed only after S1 has completed.
- We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0.
- In process P1, we insert the statements

```
S1;
signal(synch);
```

- In process P2, we insert the statements

```
wait(synch);
S2;
```

- To **overcome the problem of busy waiting**, we can modify the definition of the wait() and signal() operations as follows:
  - When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can suspend itself
  - The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state
  - Then control is transferred to the CPU scheduler, which selects another process to execute

- We define a semaphore as follows:

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

-
- Each semaphore has an integer value and a list of processes list.
- A signal() operation removes one process from the list of waiting processes
- **wait() semaphore operation**

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
              add this process to S->list;
              sleep();
        }
}
```

-
- **signal() semaphore operation**

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Monitors

- Suppose that a program interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution

```
signal(mutex);
   ...
  critical section
   ...
wait(mutex);
```

-
- In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.
- This error may be discovered only if several processes aresimultaneouslyactive in their critical sections
- Suppose that a program replaces signal(mutex) with wait(mutex).That is, it executes

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

- ☐
- ☐ In this case, the process will permanently block on the second call to wait(), as the semaphore is now unavailable
- ☐ These examples illustrate that various types of errors can be generated easily when programmers use semaphores or mutex locks incorrectly to solve the critical-section problem
- ☐ A monitor type is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor

```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
     . . .
  }

  function P2 ( . . . ) {
     . . .
  }

          .
          .
          .
  function Pn ( . . . ) {
     . . .
  }

  initialization_code ( . . . ) {
     . . .
  }
}
```

Figure 6.11 Pseudocode syntax of a monitor.

- ☐
- ☐ a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters
- ☐ Similarly, the local variables of a monitor can be accessed by only the local functions

- The monitor construct ensures that only one process at a time is active within the monitor.
- A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition: condition x, y;

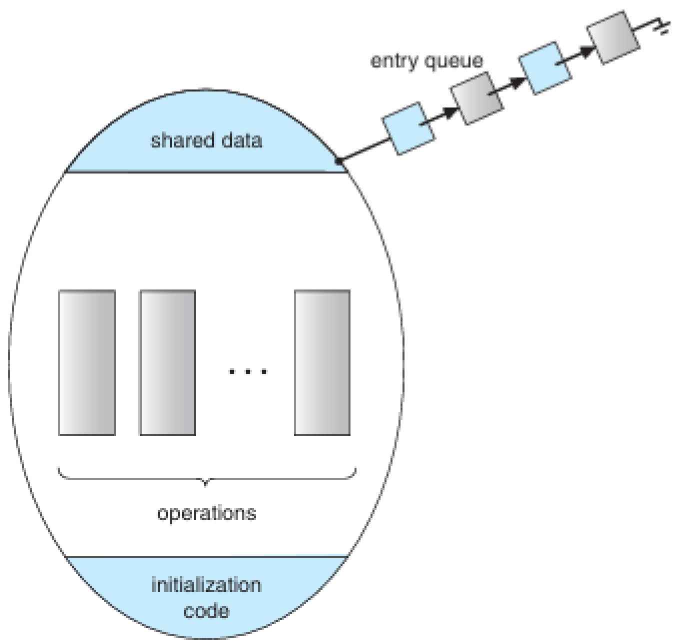
The only operations that can be invoked on a condition variable are `wait()` and `signal()`. The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

-



- **Figure 6.12** Schematic view of a monitor.
-

Classic Problems of Synchronization

The Bounded-Buffer problem

- A **producer process** produces information that is consumed by a **consumer process**
- For example, a compiler may produce assembly code that is consumed by an assembler
- The assembler, in turn, may produce object modules that are consumed by the loader

- One solution to the producer–consumer problem uses shared memory
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced
- Two types of buffers can be used
    - The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items
    - The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full
- The producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

- We assume that the pool consists of n buffers, each capable of holding one item.
- The mutex binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1
- The empty and full semaphores count the number of empty and full buffers
- The semaphore empty is initialized to the value n
- The semaphore full is initialized to the value 0.

- **Producer Process**

```
while (true) {
    . . .
  /* produce an item in next_produced */
    . . .
  wait(empty);
  wait(mutex);
    . . .
  /* add next_produced to the buffer */
    . . .
  signal(mutex);
  signal(full);
}
```

- **Figure 7.1** The structure of the producer process.
- **Consumer Process**

```
while (true) {
  wait(full);
  wait(mutex);
    . . .
  /* remove an item from buffer to next_consumed */
    . . .
  signal(mutex);
  signal(empty);
    . . .
  /* consume the item in next_consumed */
    . . .
}
```

- **Figure 7.2** The structure of the consumer process.
-

The Readers –Writers Problem

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database (readers)
- Others may want to update (that is, to read and write) the database.(writers)
- if two readers access the shared data simultaneously, no adverse effects will result.

- if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may occur
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.
- This synchronization problem is referred to as the readers–writers problem.
- **Variations**
- **first readers–writers problem**
- requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object**.**
- **The second readers –writers problem requires**
- if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in **starvation**. In the first case, writers may starve; in the second case, readers may starve.
- In the solution to the first readers–writers problem, the reader processes share the following data structures:
-

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

- The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0. The semaphore rw mutex is common to both reader and writer processes.

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

-

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

☐

☐ **The Dining-Philosophers Problem**

☐ Consider five philosophers who spend their lives thinking and eating.

☐ The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

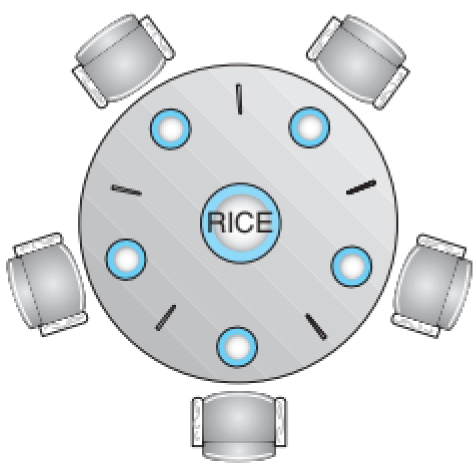


Figure 5.13 The situation of the dining philosophers.

☐

☐ In the center of the table is a bowl of rice, and the table is laid with five single chopsticks

☐ When a philosopher thinks, she does not interact with her colleagues.

☐ A philosopher may pick up only one chopstick at a time.

- She cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks.
- When she is finished eating, she puts down both chopsticks and starts thinking again.
- It is an example of a large class of concurrency-control problems.
- It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- **Solution:**
- One simple solution is to represent each chopstick with a semaphore.
- A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
- She releases her chopsticks by executing the signal() operation on the appropriate semaphores.
- The shared data are **semaphore chopstick[5];**

5.8

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Figure 5.14 The structure of philosopher *i*.

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Several possible remedies to the deadlock problem are replaced by:

- ☐ 1. Allow at most four philosophers to be sitting simultaneously at the table.
- ☐ 2. Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- ☐ 3. Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.
- ☐

☐ Deadlocks

- ☐ In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock.
- ☐ The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.
- ☐ A process must request a resource before using it and must release the resource after using it.
- ☐ A process may utilize a resource in only the following sequence:
    - ☐ 1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
    - ☐ 2. Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
    - ☐ 3. Release. The process releases the resource.
- ☐ **Necessary Conditions**
- ☐ A deadlock situation can arise if the following four conditions hold simultaneously in a system:
    - ☐ 1.**Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

- 2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
- 3. **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- 4. **Circular wait.** A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

- **Resource-Allocation Graph**
- Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.
- This graph consists of a set of vertices V and a set of edges E.
- The set of vertices V is partitioned into two different types of nodes: P = {P1, P2, ., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.
- A directed edge from process Pih to resource type Rj is denoted by Pi → Rj ; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.**(Request edge)**
- A directed edge from resource type Rj to process Pi is denoted by Rj → Pi ; it signifies that an instance of resource type Rj has been allocated to process Pi **(Assignment edge)**
- Pictorially, we represent each process Pi as a circle and each resource type Rj as a rectangle.

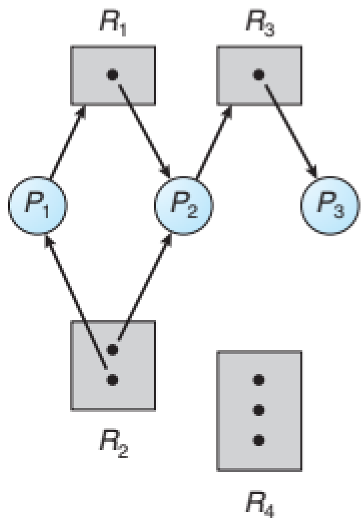


Figure 7.1 Resource-allocation graph.

-
- Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle.
- When process Pi requests an instance of resource type Rj , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.
- The resource-allocation graph shown in Figure 7.1 depicts the following situation.
- **The sets P, R, and E**:
  - P = {P1, P2, P3}
  - R = {R1, R2, R3, R4}
  - E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}
- **Resource instances**:
  - One instance of resource type R1
  - Two instances of resource type R2
  - One instance of resource type R3
  - Three instances of resource type R4
- **Process states:**
  - Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.

- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.

- If the graph contains **no cycles**, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

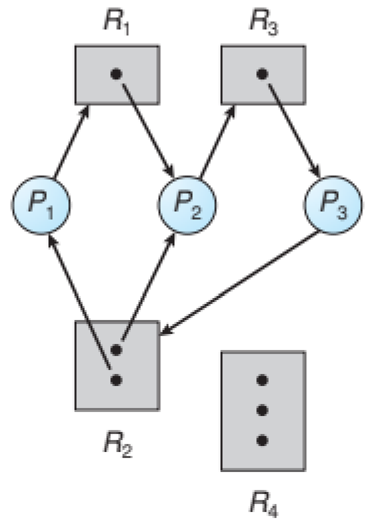


Figure 7.2 Resource-allocation graph with a deadlock.

- Suppose that process P3 requests an instance of resource, type R2. Since no resource instance is currently available, we add a request edge P3 → R2 to the graph (Figure 7.2). At this point, two minimal cycles exist in the system:
  - P1 → R1 → P2 → R3 → P3 → R2 → P1
  - P2 → R3 → P3 → R2 → P2
- Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

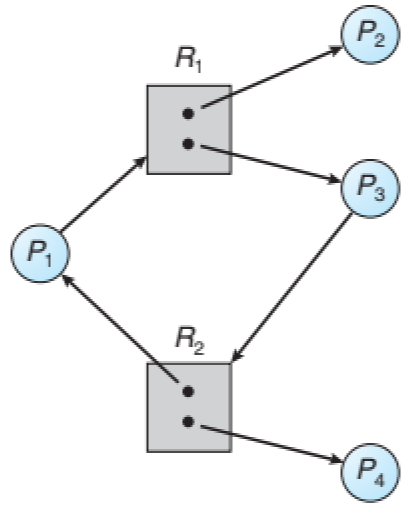


Figure 7.3 Resource-allocation graph with a cycle but no deadlock.

-
- **Methods for Handling Deadlocks**
- We can deal with the deadlock problem in one of three ways:
- 1. We can use a protocol to **prevent or avoid deadlocks**, ensuring that the system will never enter a deadlocked state.
- 2. We can allow the system to enter a deadlocked state, detect it, and **recover**.
- 3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.
- The third solution is the one used by most operating systems, including Linux and Windows. It is then up to the application developer to write programs that handle deadlocks.
- **Deadlock Prevention**
- For a deadlock to occur, each of the four necessary conditions must hold.
- By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.
- **1. Mutual Exclusion**
- The mutual exclusion condition must hold.
- That is, at least one resource **must be non sharable.**

- Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- **2. Hold and Wait**
- we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that we can use requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process to request resources only when it has none.
- Both these protocols have two main disadvantages.
    - First, resource utilization may be low, since resources may be allocated but unused for a long period.
    - Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely,
- **3. No Preemption**
- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- We can use the following protocol:
    - 1. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.
    - 2. Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they

are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait.

- ☐ **4. Circular Wait**
- ☐ One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- ☐ we let R = {R1, R2, ..., Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- ☐ We define a one-to-one function F: R → N, where N is the set of natural numbers.
- ☐ function F might be defined as follows:
- ☐ F(tape drive) = 1
- ☐ F(disk drive) = 5
- ☐ F(printer) = 12
- ☐ A process can initially request any number of instances of a resource type —say, Ri . After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri).

- ☐ **Deadlock Avoidance**
- ☐ prevent deadlocks by limiting how requests can be made.
- ☐ The simplest and most useful model requires

☐ that each process declare the maximum number of resources of each type that it may need.

☐ that ensures that the system will never enter a deadlocked state.

☐ **Safe State**

☐ A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.

☐ A system is in a safe state only if there exists a safe sequence.

☐ A sequence of processes

☐ <P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi , the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj , with j < i.

☐ An unsafe state may lead to a deadlock.

☐ **Resource-Allocation-Graph Algorithm**

☐ In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge.

☐ A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.

☐ When process Pi requests resource Rj , the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi , the Assignment edge Rj → Pi is reconverted to a claim edge Pi → Rj .

- Note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.
- suppose that process Pi requests resource Rj . The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

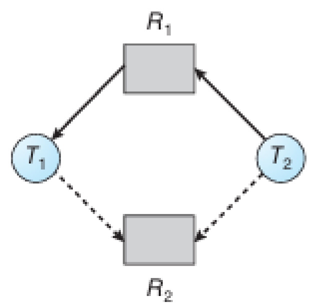


Figure 8.9 Resource-allocation graph for deadlock avoidance.

-
- To illustrate this algorithm, we consider the resource-allocation graph of Figure 8.9. Suppose that T2 requests R2.AlthoughR2 is currently free, we cannot allocate it to T2, since this action will create a cycle in the graph (Figure 8.10). A cycle, as mentioned, indicates that the system is in an unsafe state. If T1 requests R2,andT2 requests R1, then a deadlock will occur

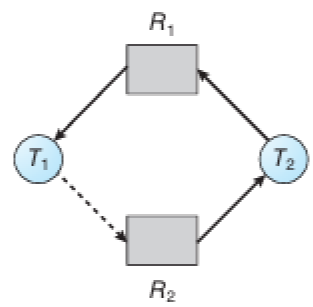


Figure 8.10 An unsafe state in a resource-allocation graph.

-

Banker's Algorithm

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- **Data structures** must be maintained to implement the Banker's algorithm.
  - **Available**. A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.
  - **Max.** An n × m matrix defines the maximum demand of each process. If Max[i][j] equals k, then process Pi may request at most k instances of resource type Rj
  - **Allocation**. An n × m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj

- ☐ **Need**. An n × m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task. Note that Need[i][j] equals Max[i][j] - Allocation[i][j].

☐ **Safety Algorithm**

☐ 1.Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1.

☐ 2. Find an index i such that both

- ☐ a. Finish[i] == false
- ☐ b. Needi ≤ Work
- ☐ If no such i exists, go to step 4.

☐ 3. Work = Work + Allocationi

- ☐ 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**

☐ the algorithm for determining whether requests can be safely granted.

☐ Let Requesti be the request vector for process Pi . If Requesti [j] == k, then process Pi wants k instances of resource type Rj

☐ When a request for resources is made by process Pi , the following actions are taken:

- ☐ 1. If Requesti ≤ Needi , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
- ☐ 2. If Requesti ≤ Available, go to step 3. Otherwise, Pi must wait, since the resources are not available.
- ☐ 3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
  - ☐ Available = Available−Requesti ;
  - ☐ Allocationi = Allocationi + Requesti ;
  - ☐ Needi = Needi −Requesti ;

- ☐ If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources. However, if the new state is unsafe, then Pi must wait for Requesti , and the old resource-allocation state is restored.
- ☐ Example:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- ☐
- ☐ Content of matrix need:

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- ☐
- ☐ We claim that the system is currently in a safe state. Indeed, the sequence <P1, P3, P4, P2, P0> satisfies the safety criteria.
- ☐ Suppose now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2).
- ☐ To decide whether this request can be immediately granted, we first check that Request1 ≤ Available—that is, that (1,0,2) ≤ (3,3,2), which is true.
- ☐ We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <P1, P3, P4, P0, P2> satisfies the safety requirement. Hence, we can immediately grant the request of process P1.
- when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available.

Memory Management

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes.
- The memory is usually divided into two partitions:
  - one for the resident operating system and
  - one for the user processes.
- We can place the operating system in either low memory or high memory.
  - The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- We usually want several user processes to reside in memory at the same time.
- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

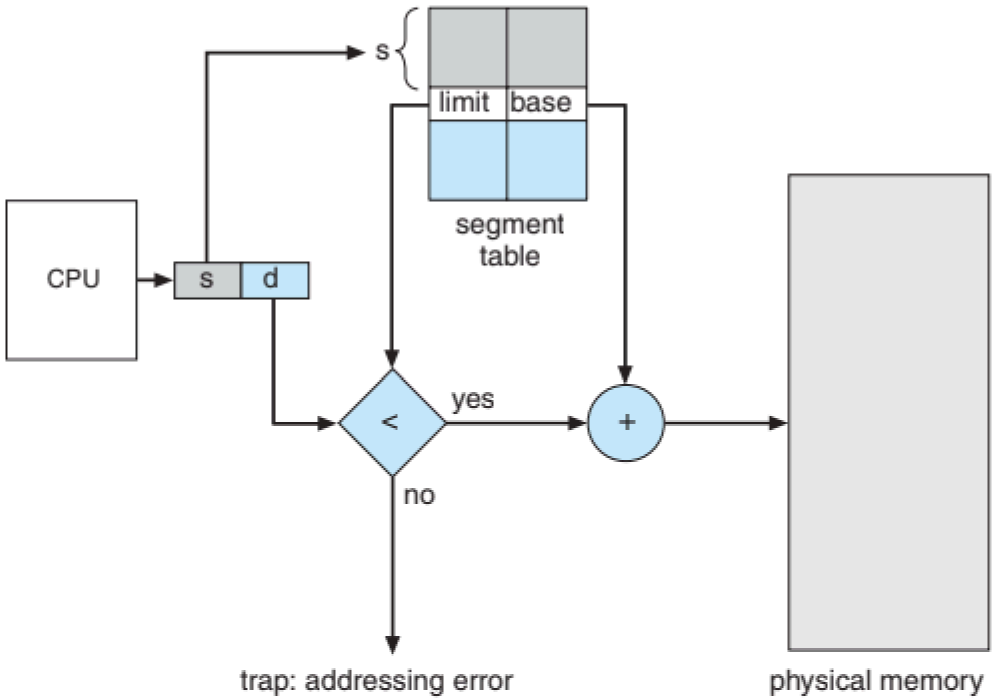
- ☐ One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**.
  - ☐ Each partition may contain exactly one process.
  - ☐ Thus, the degree of multiprogramming is bound by the number of partitions.
  - ☐ In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- ☐ In the **variable-partition scheme**, the operating system keeps a table indicating which parts of memory are available and which are occupied.
- ☐ Initially, all memory is available for user processes and is considered one large block of available memory, a hole.
- ☐ Eventually, as you will see, memory contains a set of holes of various sizes.
- ☐ As processes enter the system, they are put into an input queue. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- ☐ The memory blocks available comprise a set of holes of various sizes scattered throughout memory.
- ☐ When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- ☐ If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.
- ☐ When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- ☐ **dynamic storage-allocation problem** - concerns how to satisfy a request of size n from a list of free holes.
- ☐ Solutions to this problem
- ☐ **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the

previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

- **Best fit.** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.
- Problem discussed in class regarding first fit, best fit and worst fit strategies
- **Memory fragmentation** can be internal as well as external.
-
- **Segmentation**
- When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions.
- It may also include various data structures: objects, arrays, stacks, variables, and so on
- Each of these modules or data elements is referred to by name.
- The programmer talks about “the stack,” “the math library,” and “the main program” without caring what addresses in memory these elements occupy.
- Segments vary in length, and the length of each is intrinsically defined by its purpose in the program.
- Segmentation is a memory-management scheme that supports this programmer view of memory.
- A logical address space is a collection of segments.
- Each segment has a name and a length.

☐ The addresses specify both the segment name and the offset within the segment.
☐ The programmer therefore specifies each address by two quantities: a segment name and an offset.
☐ A logical address: <segment-number, offset>.
☐ A C compiler might create separate segments for the following:
☐ 1. The code
☐ 2. Global variables
☐ 3. The heap, from which memory is allocated
☐ 4. The stacks used by each thread
☐ 5. The standard C library
☐ Libraries that are linked in during compile time might be assigned separate segments.
☐ The loader would take all these segments and assign them segment numbers.
☐ **Segmentation Hardware**
☐ Each entry in the segment table has a segment base and a segment limit.
☐ The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.
☐ A logical address consists of two parts: a segment number, s, and an offset into that segment, d.
☐ The segment number is used as an index to the segment table
☐ The offset d of the logical address must be between 0 and the segment limit.

☐



☐ **Figure 8.8** Segmentation hardware.

☐ Example of Segmentation

☐ We have five segments numbered from 0 through 4.

☐ The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).

☐ a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.

☐ A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.

☐ A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

☐

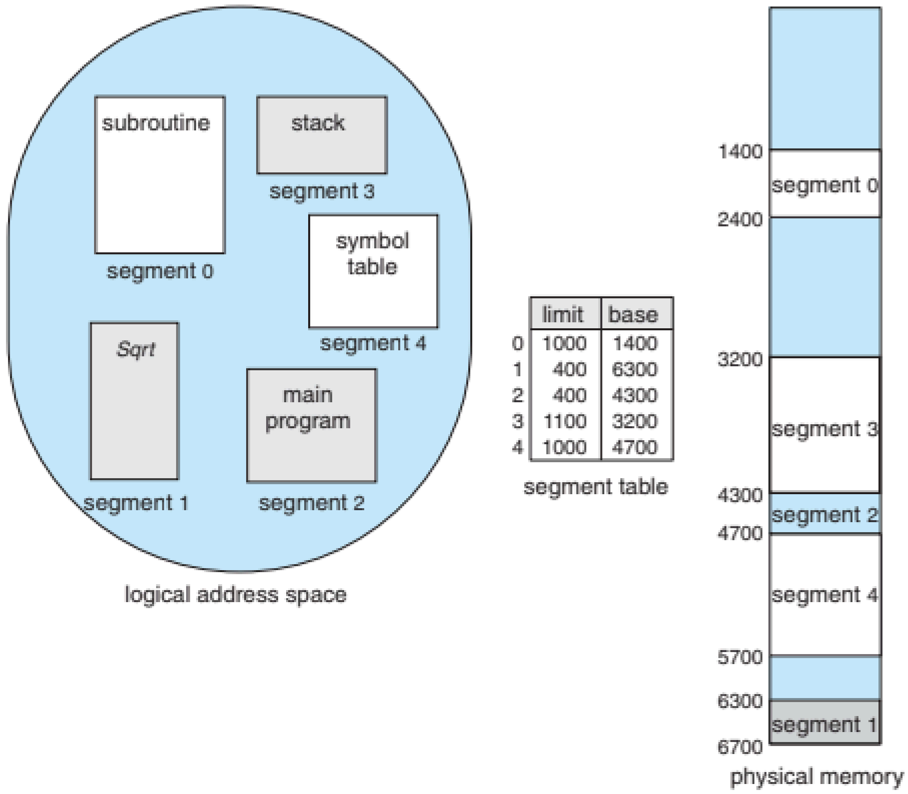


Figure 8.9 Example of segmentation.