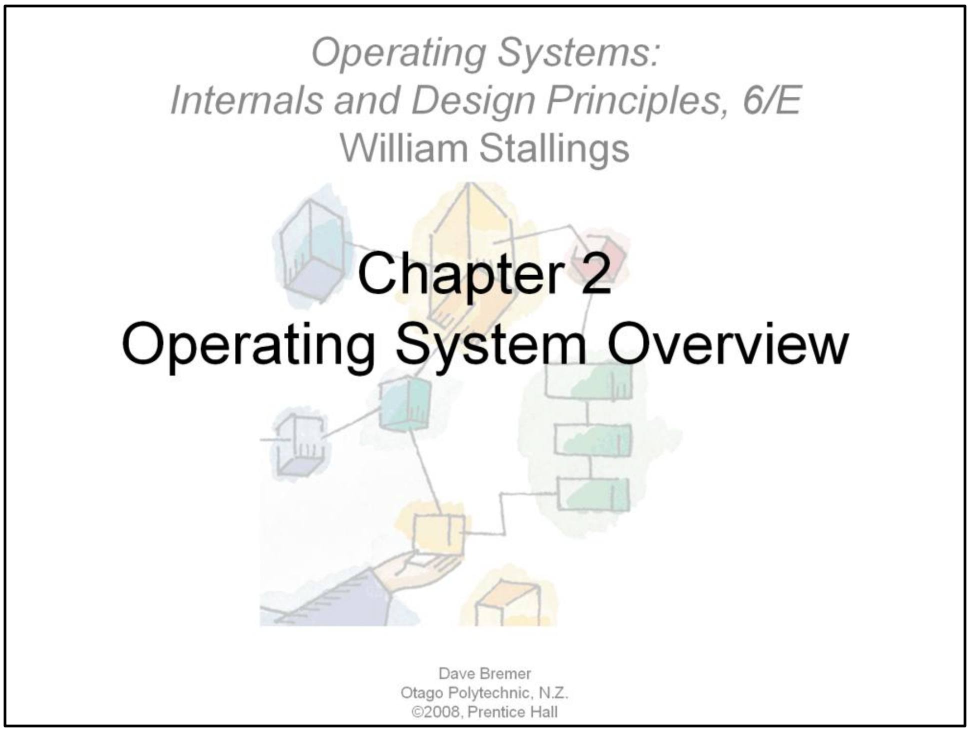
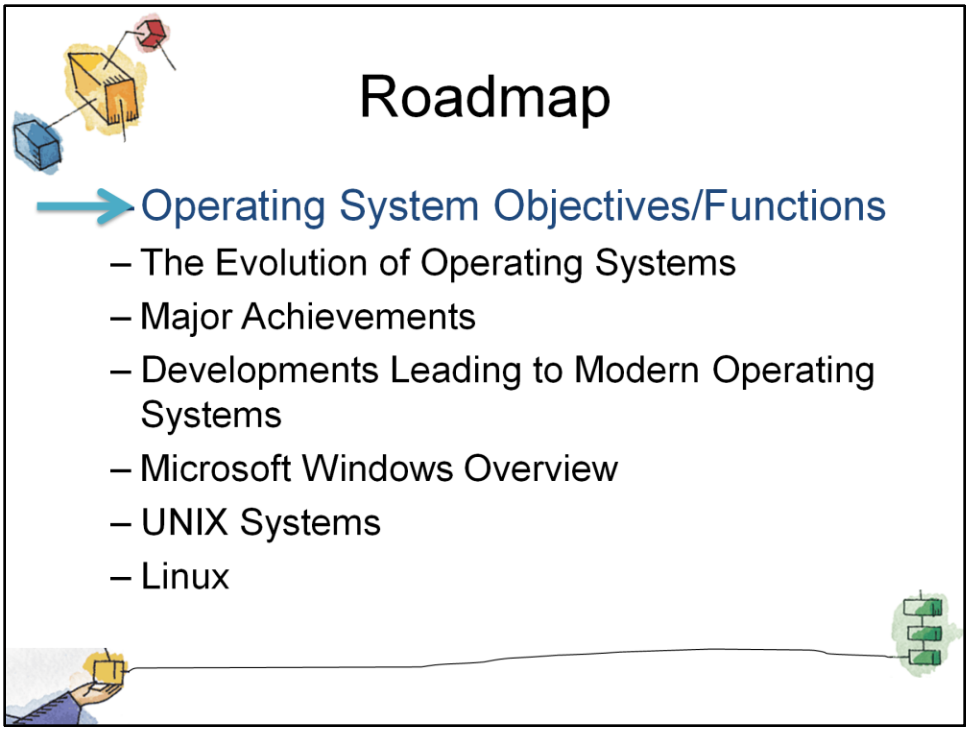


Operating Systems:
Internals and Design Principles, 6/E
William Stallings

Chapter 2 Operating System Overview

Dave Bremer
Otago Polytechnic, N.Z.

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.



Roadmap

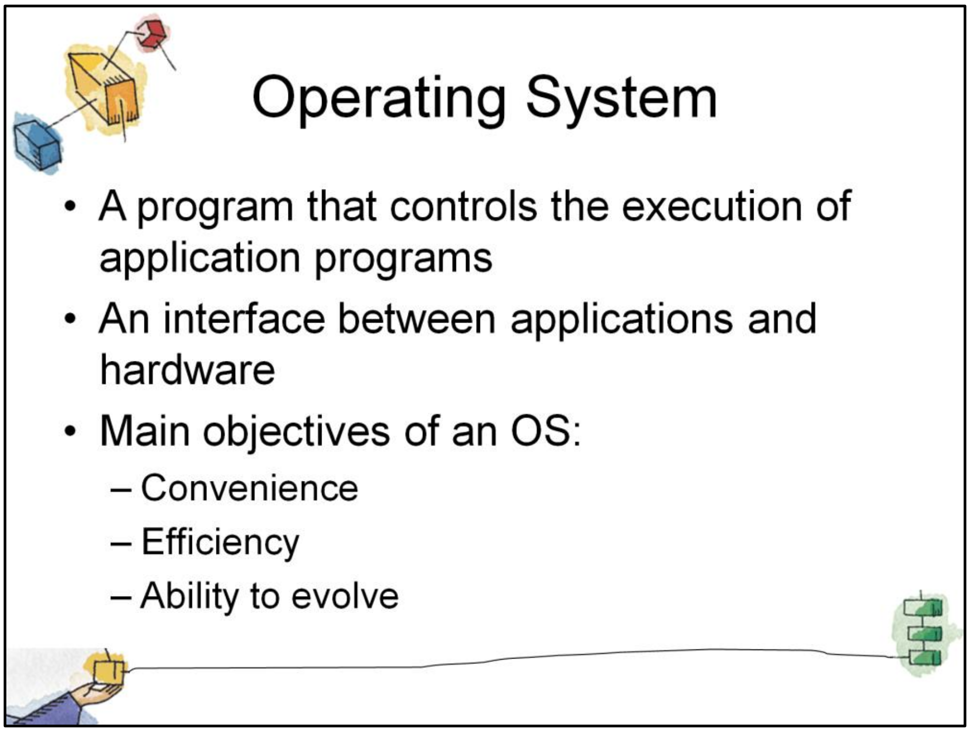
- Operating System Objectives/Functions
  - The Evolution of Operating Systems
  - Major Achievements
  - Developments Leading to Modern Operating Systems
  - Microsoft Windows Overview
  - UNIX Systems
  - Linux

The first section examines the objectives and functions of operating systems.

Then we look at how operating systems have evolved from primitive batch systems to sophisticated multitasking, multiuser systems.

The remainder of the chapter looks at the history and general characteristics of the two operating systems that serve as examples throughout this book.

NOTE: All of the material in this chapter is covered in greater depth later in the book.



Operating System

- A program that controls the execution of application programs
- An interface between applications and hardware
- Main objectives of an OS:
  - Convenience
  - Efficiency
  - Ability to evolve

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware.

It can be thought of as having three objectives:

Convenience:

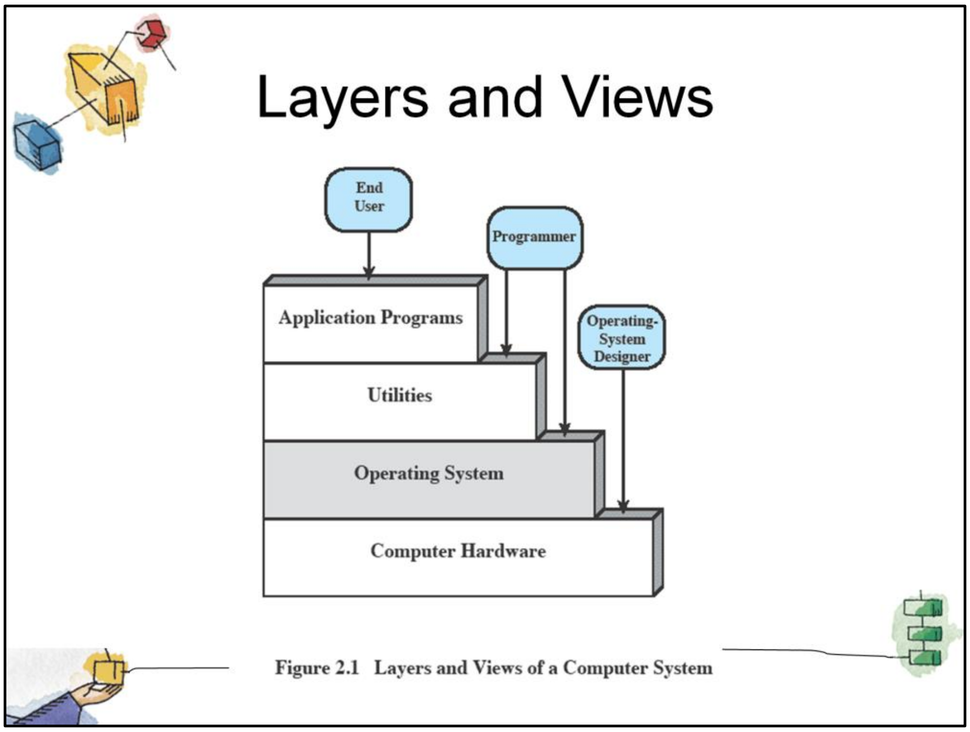
• An OS makes a computer more convenient to use.

Efficiency:

• An OS allows the computer system resources to be used in an efficient manner.

Ability to evolve:

• An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.



Layers and Views

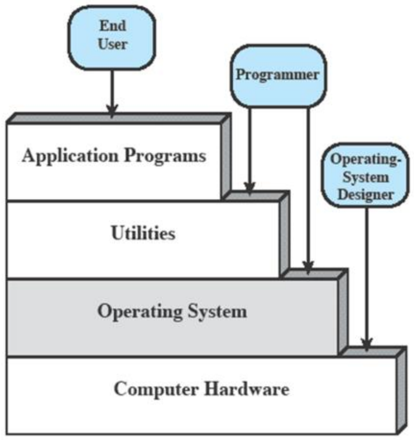


Figure 2.1 Layers and Views of a Computer System

The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted here.

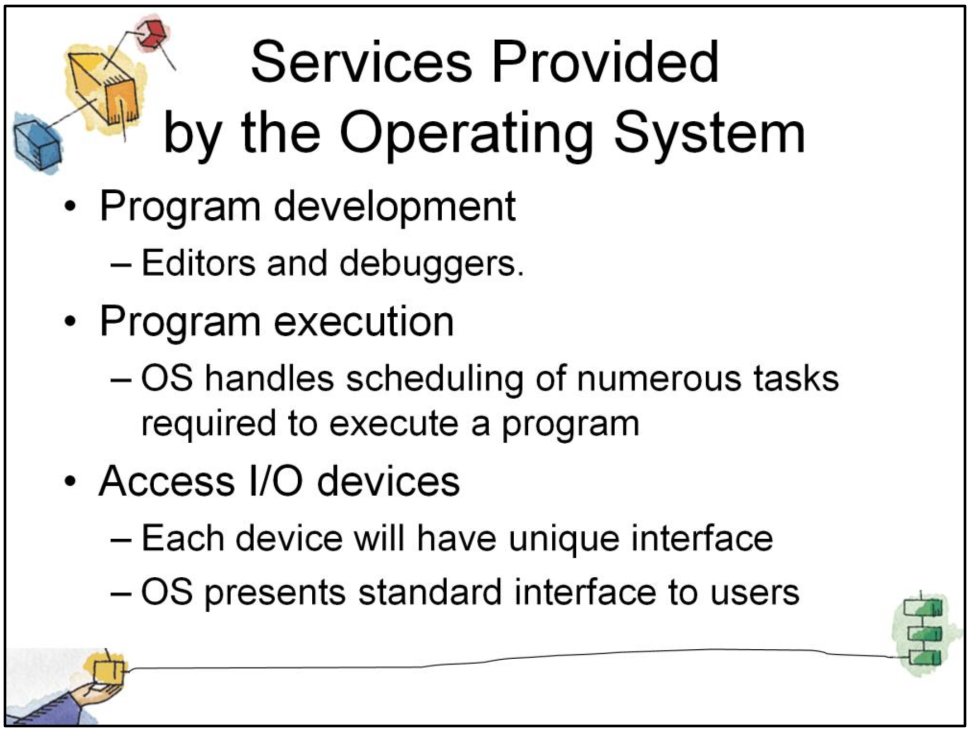
The user of those applications, the end user, generally is not concerned with the details of computer hardware.

- The end user views a computer system in terms of a set of applications.
- An application can be expressed in a programming language and is developed by an application programmer.

If one were to develop an application program as a set of machine instructions that is completely responsible for controlling the computer hardware, one would be faced with an overwhelmingly complex undertaking.

The most important collection of system programs comprises the OS.

- The OS masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system.
- It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.



Services Provided by the Operating System

- Program development
  - Editors and debuggers.
- Program execution
  - OS handles scheduling of numerous tasks required to execute a program
- Access I/O devices
  - Each device will have unique interface
  - OS presents standard interface to users

Program development:

• The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs.

• Typically, these services are in the form of utility programs that, while not strictly part of the core of the OS, are supplied with the OS and are referred to as application program development tools.

Program execution:

• A number of steps need to be performed to execute a program.

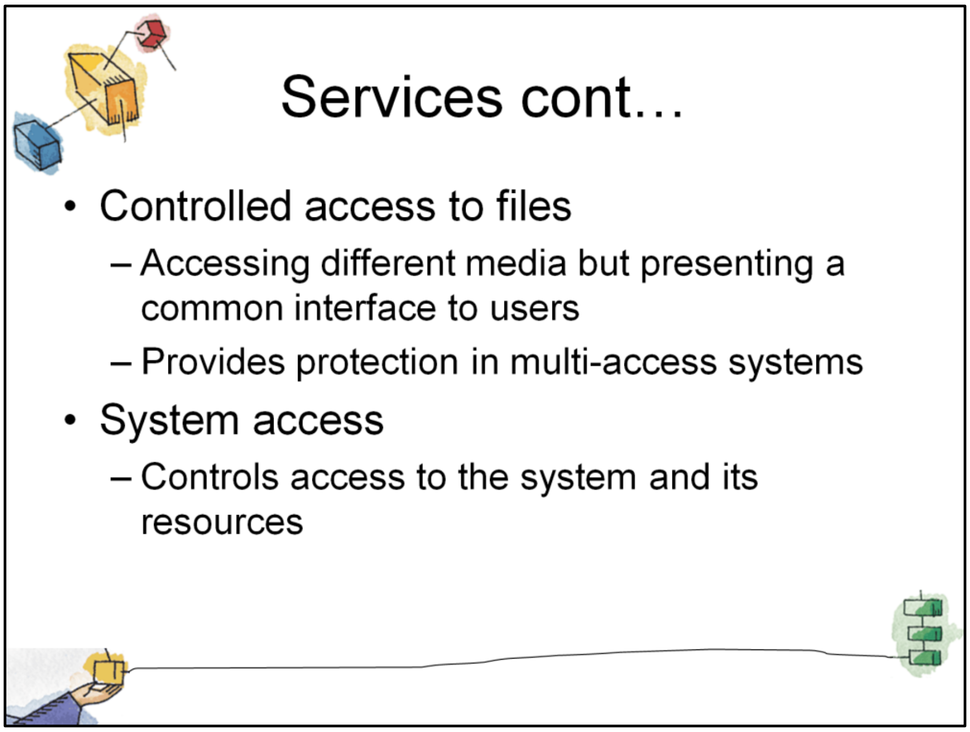
• Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared.

• The OS handles these scheduling duties for the user.

Access to I/O devices:

• Each I/O device requires its own peculiar set of instructions or control signals for operation.

• The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.



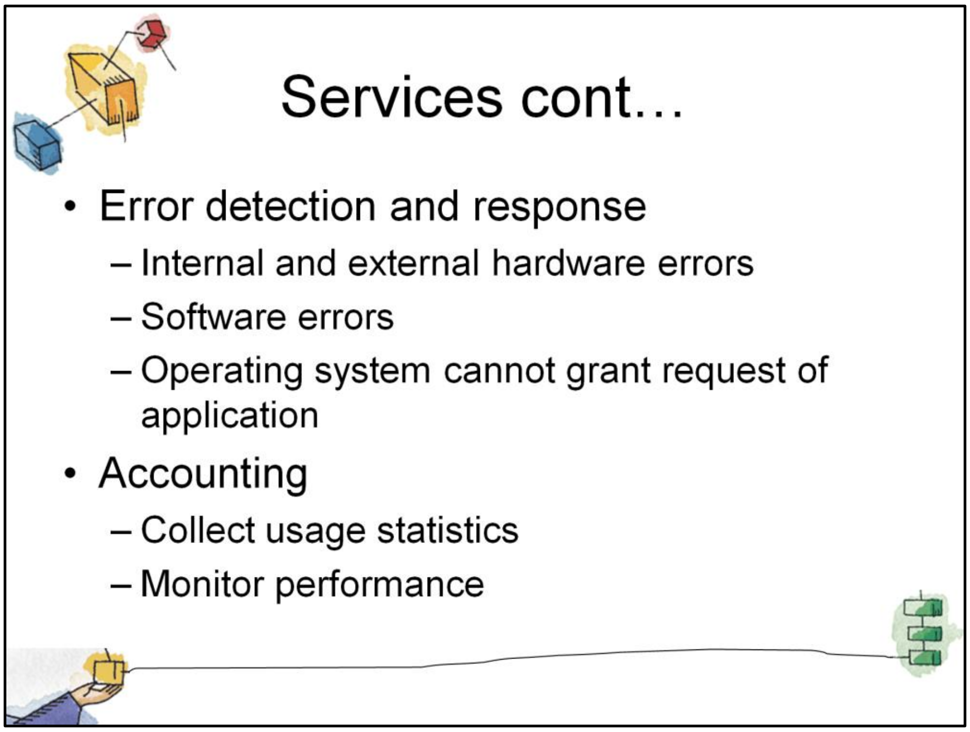
Services cont…

- Controlled access to files
  - Accessing different media but presenting a common interface to users
  - Provides protection in multi-access systems
- System access
  - Controls access to the system and its resources

Controlled access to files:

- For file access, the OS must reflect a detailed understanding of not only the nature of the I/O device (disk drive, tape drive) but also the structure of the data contained in the files on the storage medium.
- In the case of a system with multiple users, the OS may provide protection mechanisms to control access to the files.

- **System access:**
  - For shared or public systems, the OS controls access to the system as a whole and to specific system resources.
  - The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.



Services cont…

- Error detection and response
  - Internal and external hardware errors
  - Software errors
  - Operating system cannot grant request of application
- Accounting
  - Collect usage statistics
  - Monitor performance

Error detection and response:

• A variety of errors can occur while a computer system is running.

• These include internal and external hardware errors, such as a memory error, or a device failure or malfunction;

• and various software errors, such as division by zero, attempt to access forbidden memory location, and inability of the OS to grant the request of an application.

• In each case, the OS must provide a response that clears the error condition with the least impact on running applications.

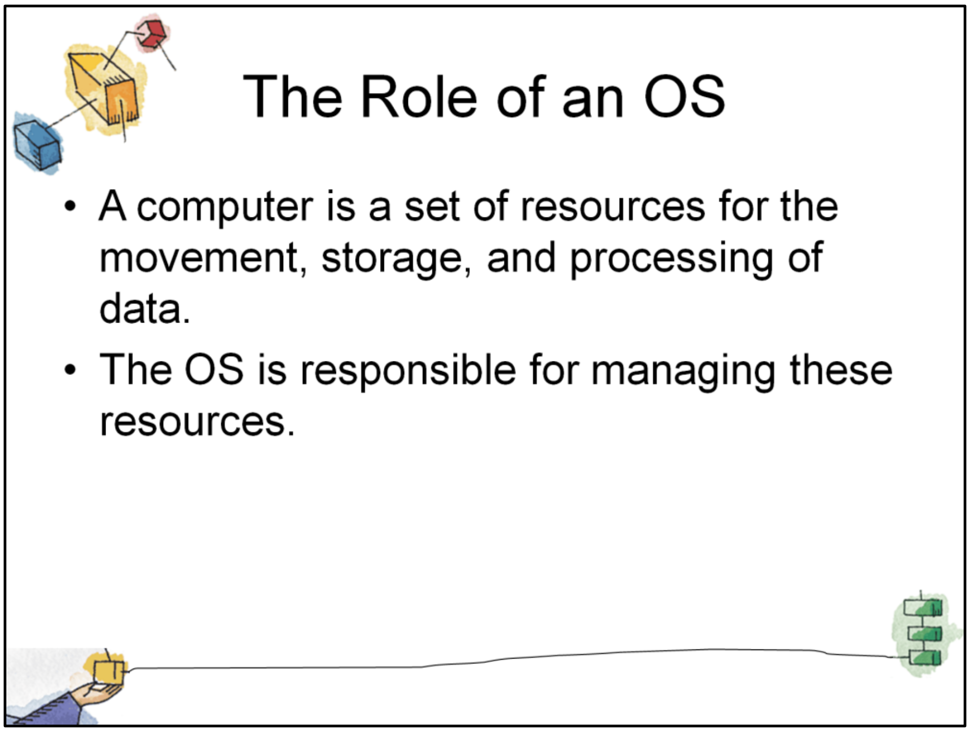
• The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.

Accounting:

• A good OS will collect usage statistics for various resources and monitor performance parameters such as response time.

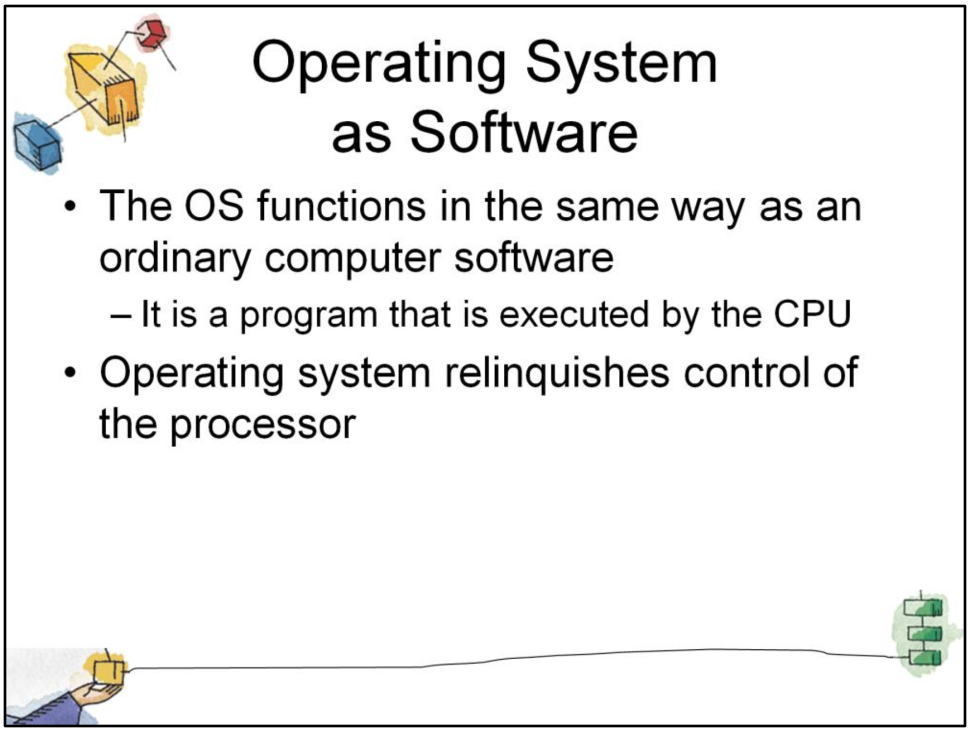
• On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance.

• On a multiuser system, the information can be used for billing purposes.



The Role of an OS

- A computer is a set of resources for the movement, storage, and processing of data.
- The OS is responsible for managing these resources.



Operating System as Software

- The OS functions in the same way as an ordinary computer software
  - It is a program that is executed by the CPU
- Operating system relinquishes control of the processor

The OS functions in the same way as ordinary computer software;

- It is a program or suite of programs executed by the processor.

The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

OS as Resource Manager

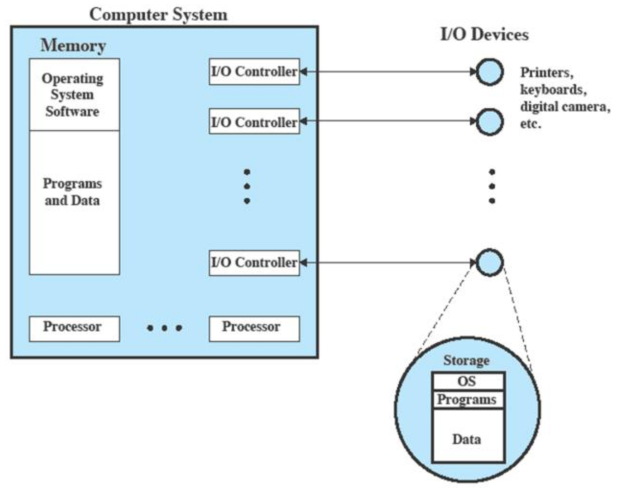


Figure 2.2 The Operating System as Resource Manager

This figure suggests the main resources that are managed by the OS.

A portion of the OS is in main memory.

- This includes the kernel, or nucleus, which contains the most frequently used functions in the OS
- and other portions of the OS currently in use.

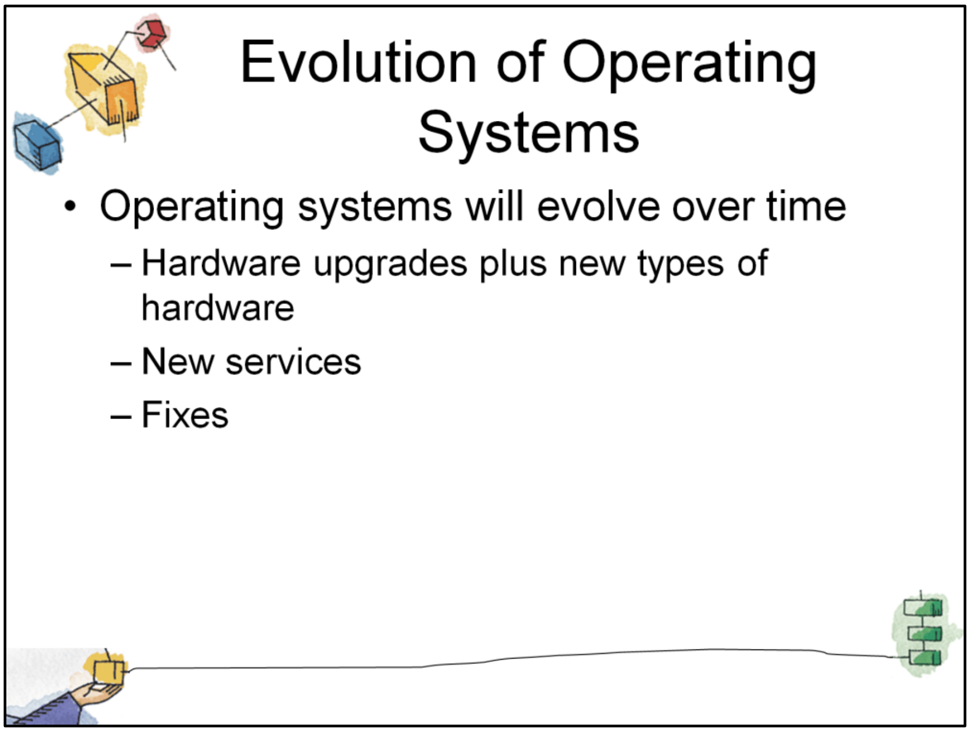
The remainder of main memory contains user programs and data.

- The allocation of this resource (main memory) is controlled jointly by the OS and memory management hardware in the processor.

The OS decides when an I/O device can be used by a program in execution and controls access to and use of files.

The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program.

- In the case of a multiple-processor system, this decision must span all of the processors.



Evolution of Operating Systems

- Operating systems will evolve over time
  - Hardware upgrades plus new types of hardware
  - New services
  - Fixes

Hardware upgrades plus new types of hardware:

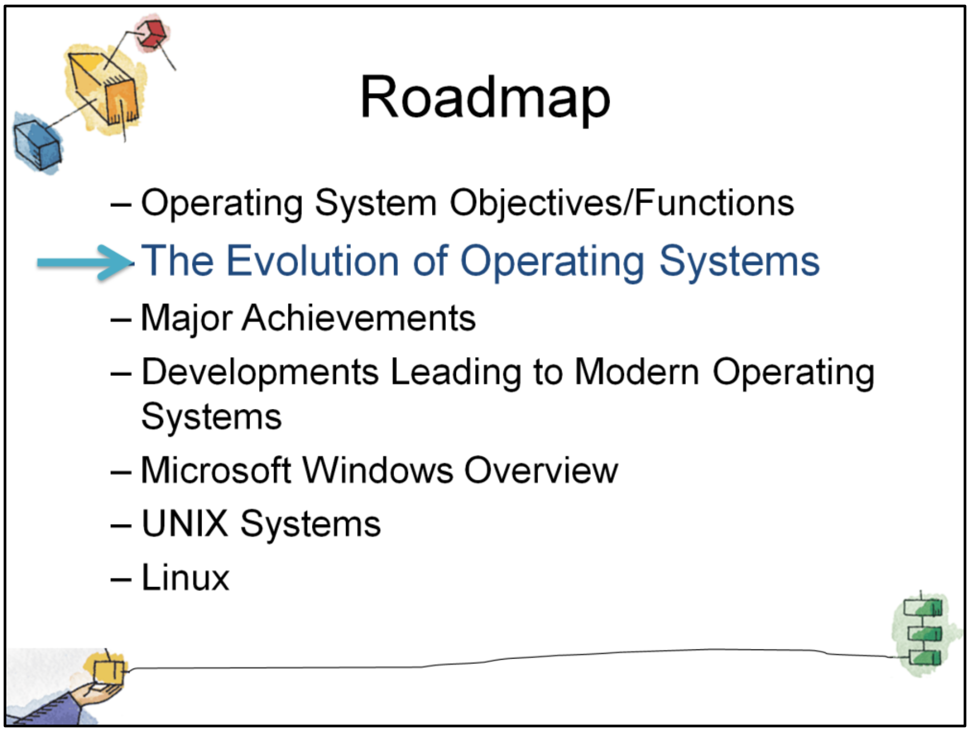
- e.g. early versions of UNIX and the Macintosh operating system did not employ a paging mechanism because they were run on processors without paging hardware.
- Subsequent versions of these operating systems were modified to exploit paging capabilities.
- Also, the use of graphics terminals and page-mode terminals instead of line-at-a-time scroll mode terminals affects OS design.
- For example, a graphics terminal typically allows the user to view several applications at the same time through "windows" on the screen.
- This requires more sophisticated support in the OS.

New services:

- In response to user demand or in response to the needs of system managers, the OS expands to offer new services.
- e.g. if it is found to be difficult to maintain good performance for users with existing tools, new measurement and control tools may be added to the OS.

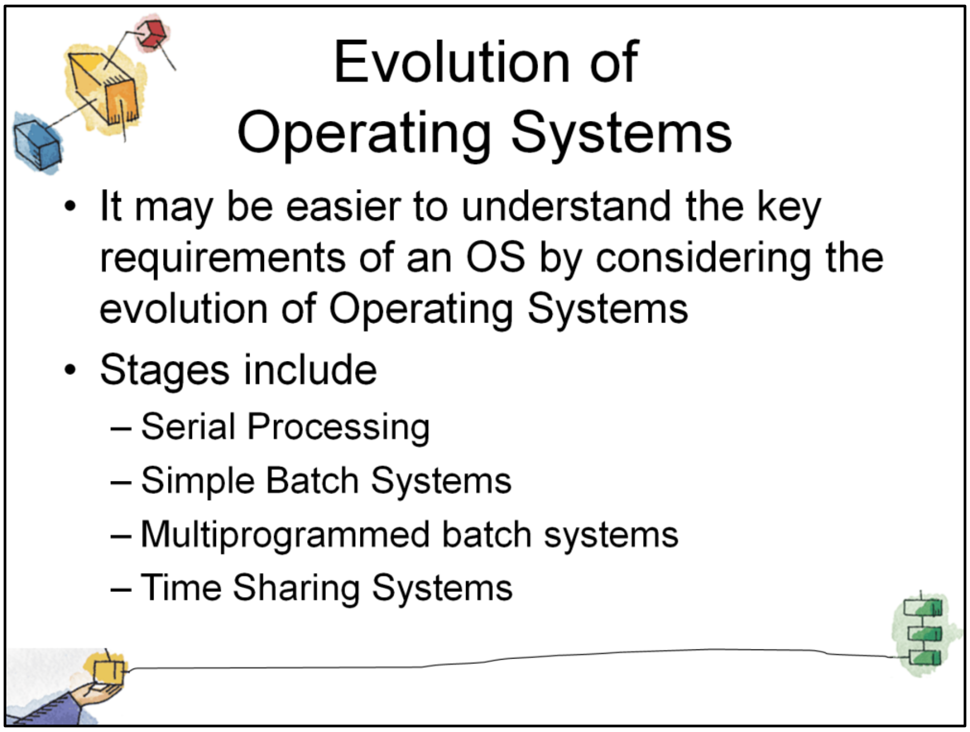
Fixes:

- Any OS has faults.
- These are discovered over the course of time and fixes are made.
- Of course, the fix may introduce new faults.



Roadmap

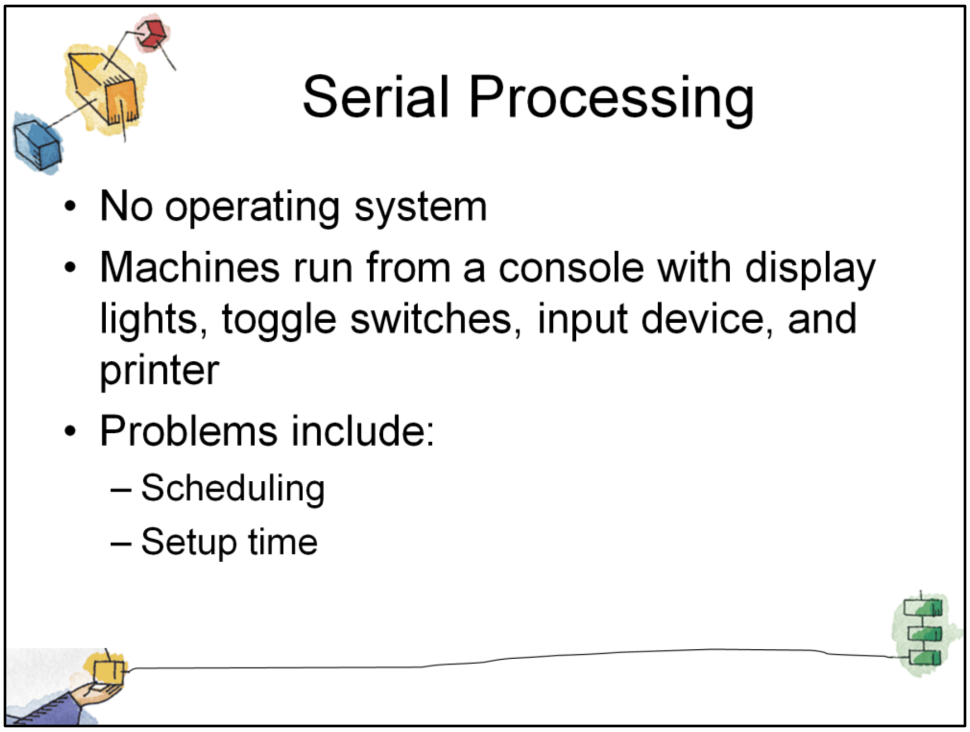
- Operating System Objectives/Functions
- → The Evolution of Operating Systems
- Major Achievements
- Developments Leading to Modern Operating Systems
- Microsoft Windows Overview
- UNIX Systems
- Linux



Evolution of Operating Systems

- It may be easier to understand the key requirements of an OS by considering the evolution of Operating Systems
- Stages include
  - Serial Processing
  - Simple Batch Systems
  - Multiprogrammed batch systems
  - Time Sharing Systems

In attempting to understand the key requirements for an OS and the significance of the major features of a contemporary OS, it is useful to consider how operating systems have evolved over the years.



Serial Processing

- No operating system
- Machines run from a console with display lights, toggle switches, input device, and printer
- Problems include:
  - Scheduling
  - Setup time

With the earliest computers (late 1940s to the mid-1950s), the programmer interacted directly with the computer hardware;

- There was no OS.
- These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer.
- 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.

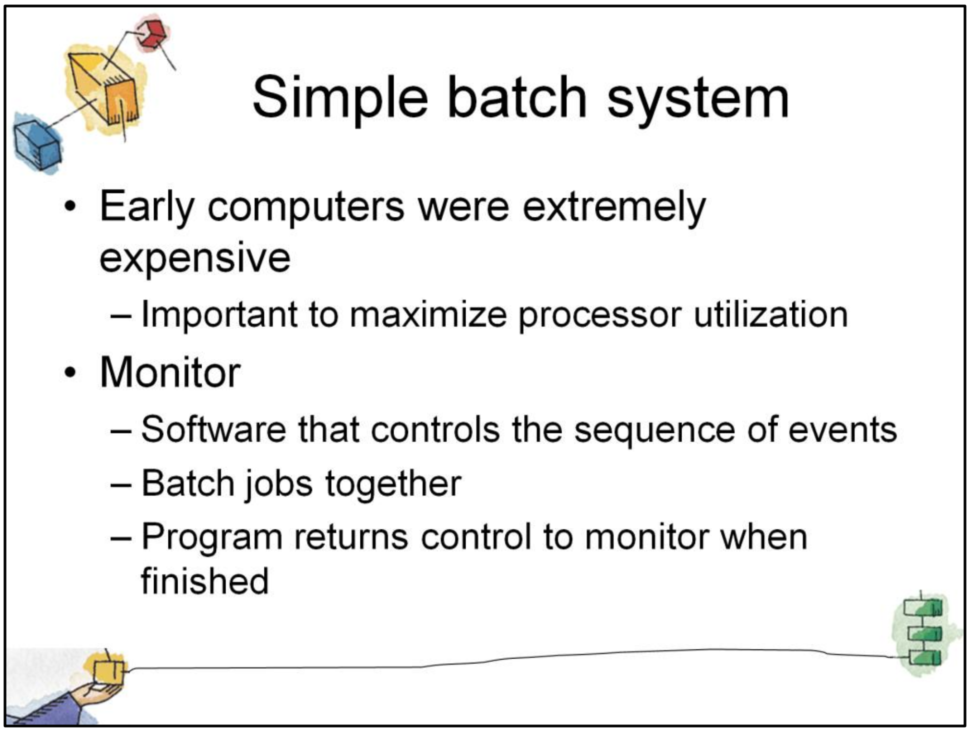
These early systems presented two main problems:

Scheduling:

- Most installations used a hardcopy sign-up sheet to reserve computer time.
- Typically, a user could sign up for a block of time in multiples of a half hour or so.
- A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer processing time.
- 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.

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) and then loading and linking together the object program and common functions.
- Each of these steps could involve mounting or dismounting tapes or setting up card decks.
- If an error occurred, the hapless user typically had to go back to the beginning of the setup sequence.
- Thus, a considerable amount of time was spent just in setting up the program to run



Simple batch system

- Early computers were extremely expensive
  - Important to maximize processor utilization
- Monitor
  - Software that controls the sequence of events
  - Batch jobs together
  - Program returns control to monitor when finished

Early computers were very expensive, and therefore it was important to maximize processor utilization.

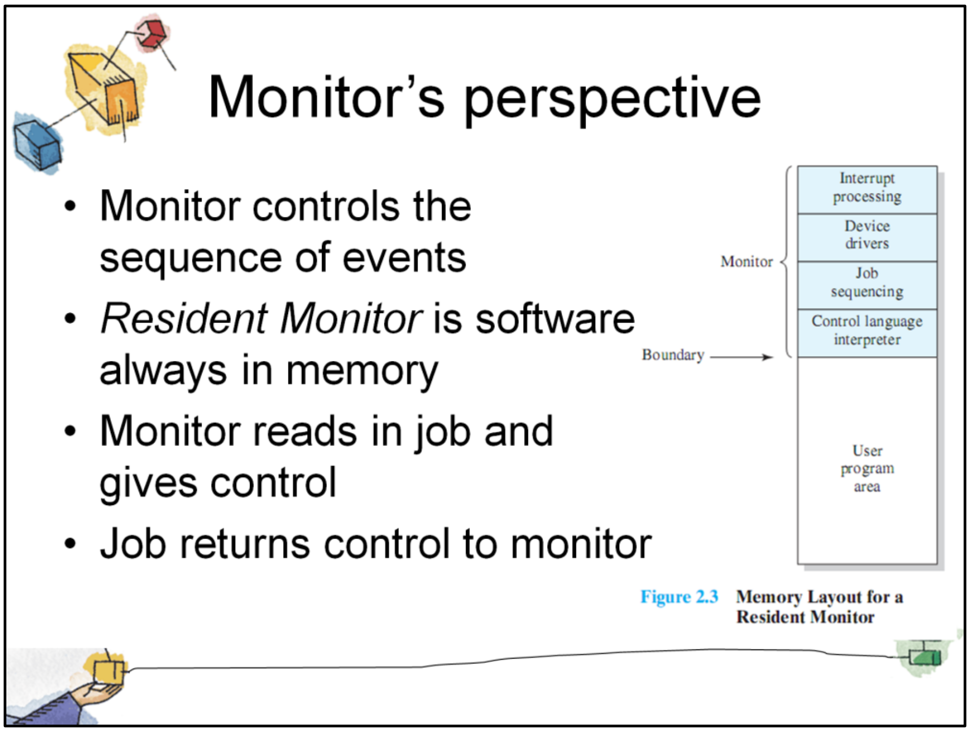
- The wasted time due to scheduling and setup time was unacceptable.

To improve utilization, the concept of a batch operating system was developed.

The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor.**

With this type of OS, **the user no longer has direct access to the processor**.

- Instead, the 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, at which point the monitor automatically begins loading the next program.



Monitor's perspective

- Monitor controls the sequence of events
- *Resident Monitor* is software always in memory
- Monitor reads in job and gives control
- Job returns control to monitor

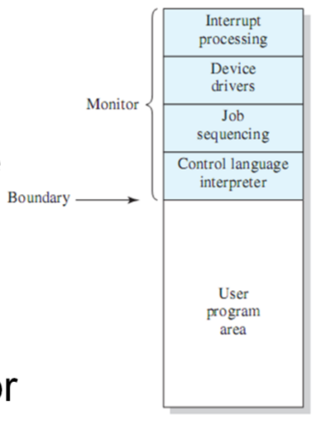


Figure 2.3 **Memory Layout for a Resident Monitor**

Monitor point of view: The monitor controls the sequence of events.

• Much of the monitor must always be in main memory and available for execution (Figure 2.3).

• That portion is referred to as the **resident monitor**.

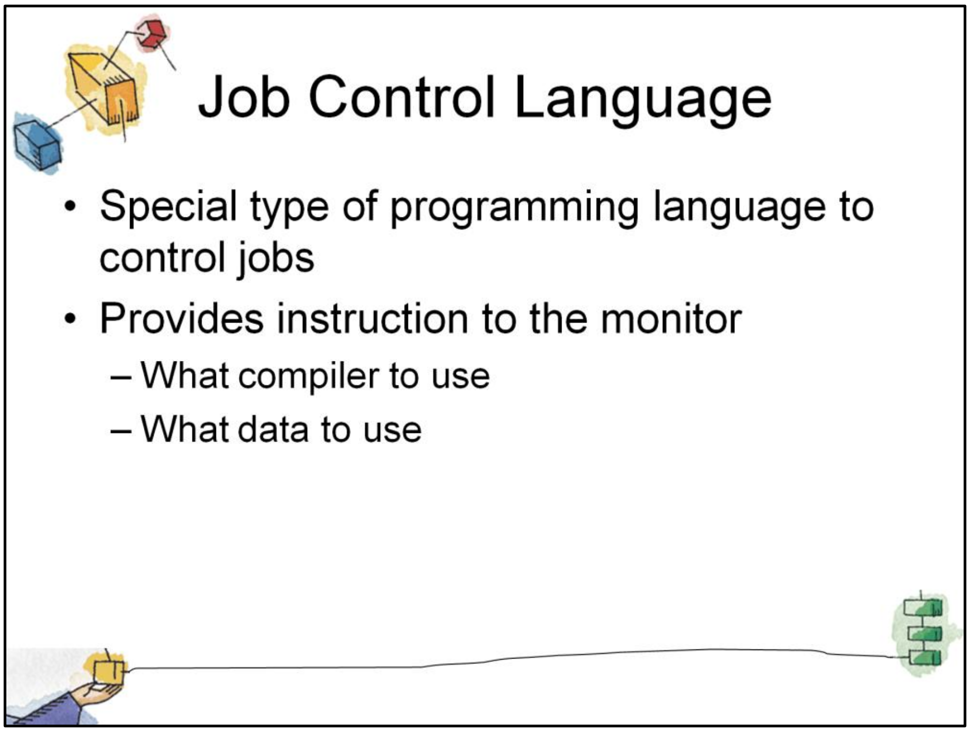
• The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them.

• 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.

• When the job is completed, it returns control to the monitor, which immediately reads in the next job.

• The results of each job are sent to an output device, such as a printer, for delivery to the user.



Job Control Language

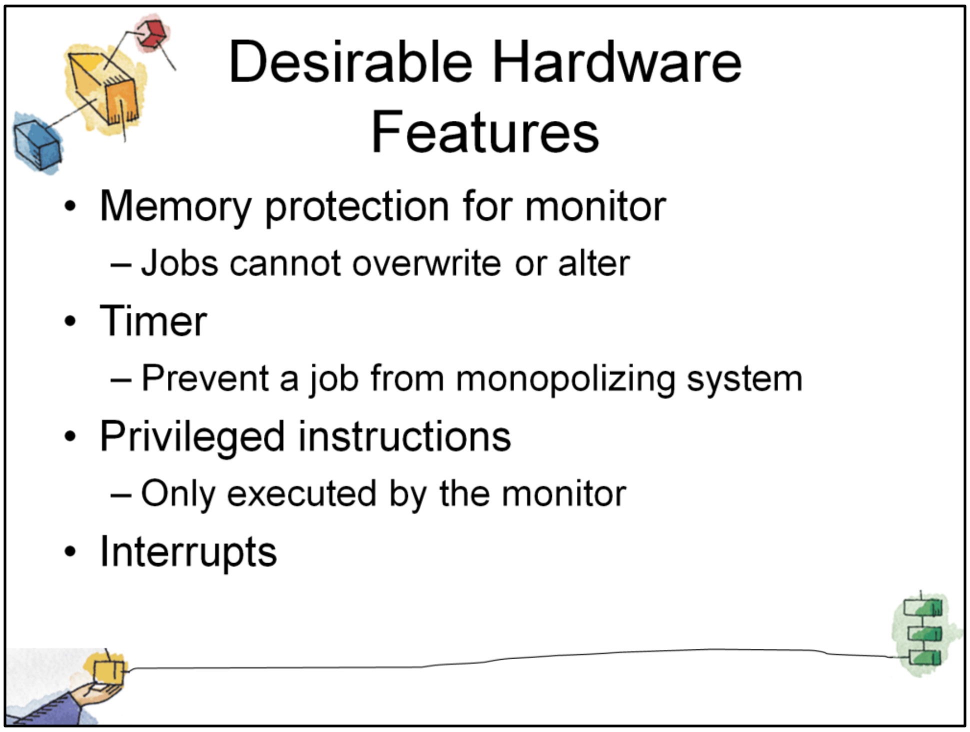
- Special type of programming language to control jobs
- Provides instruction to the monitor
  - What compiler to use
  - What data to use

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.
- The monitor improves job setup time as well.

With each job, instructions are included in a primitive form of job control language (JCL).

- A special type of programming language used to provide instructions to the monitor.



Desirable Hardware Features

- Memory protection for monitor
  - Jobs cannot overwrite or alter
- Timer
  - Prevent a job from monopolizing system
- Privileged instructions
  - Only executed by the monitor
- Interrupts

The monitor, or batch operating system, is simply a computer program.

- It relies on the ability of the processor to fetch instructions from various portions of main memory to alternately seize and relinquish control.

Certain other hardware features are also desirable:

Memory protection: While the user program is executing, it must not alter the memory area containing the monitor.

- If such an attempt is made, the processor hardware should detect an error and transfer control to the monitor.
- The monitor would then abort the job, print out an error message, and load in the next job.

Timer: A timer is used to prevent a single job from monopolizing the system.

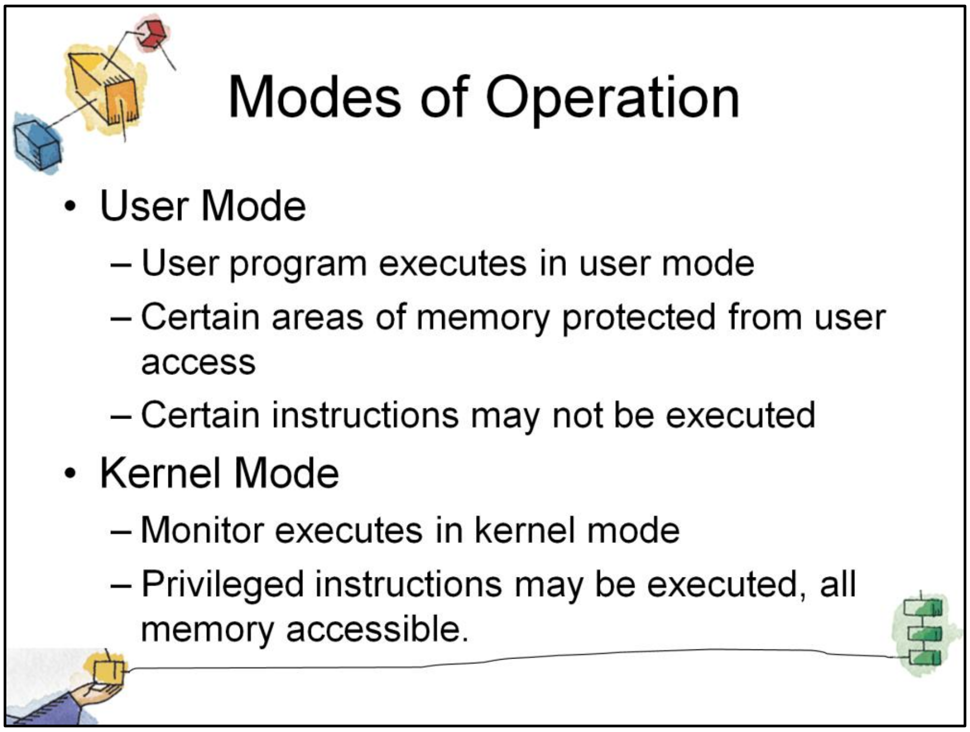
- The timer is set at the beginning of each job.
- If the timer expires, the user program is stopped, and control returns to the monitor.

Privileged instructions: Certain machine level instructions are designated privileged and can be executed only by the monitor.

- If the processor encounters such an instruction while executing a user program, an error occurs causing control to be transferred to the monitor.
- Among the privileged instructions are I/O instructions, so that the monitor retains control of all I/O devices.
- E.G. this prevents a user program from accidentally reading job control instructions from the next job.
- If a user program wishes to perform I/O, it must request that the monitor perform the operation for it.

Interrupts: Early computer models did not have this capability.

- This feature gives the OS more flexibility in relinquishing control to and regaining control from user programs.



Modes of Operation

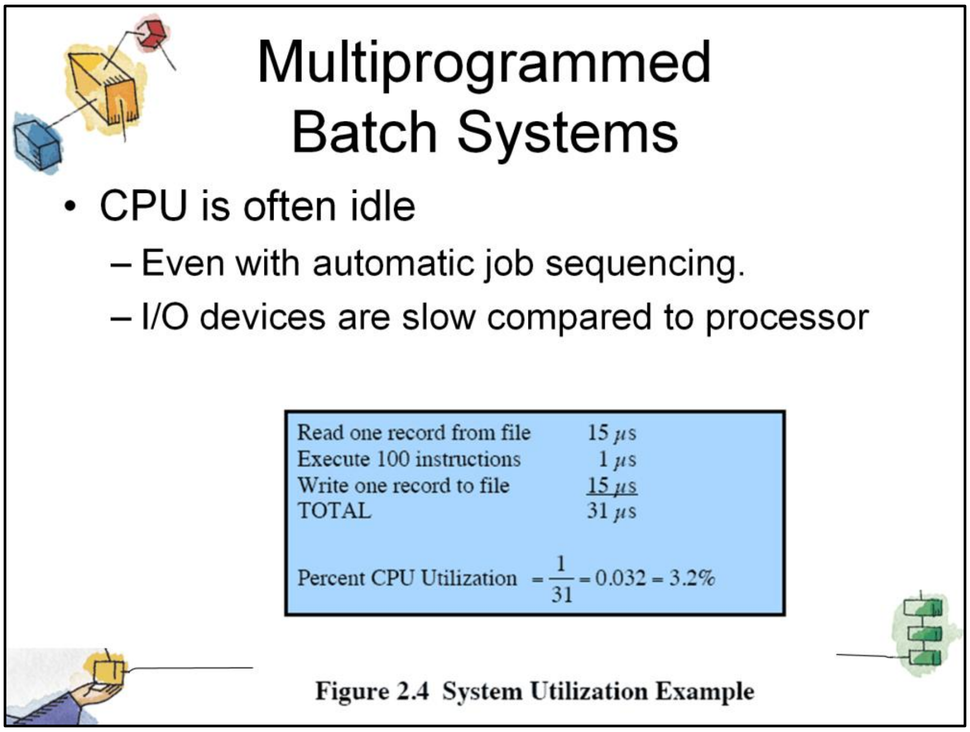
- User Mode
  - User program executes in user mode
  - Certain areas of memory protected from user access
  - Certain instructions may not be executed
- Kernel Mode
  - Monitor executes in kernel mode
  - Privileged instructions may be executed, all memory accessible.

Considerations of memory protection and privileged instructions lead to the concept of modes of operation.

A user program executes in a user mode, in which certain areas of memory are protected from the user’s use and in which certain instructions may not be executed.

The monitor executes in a system mode, or what has come to be called kernel mode, in which privileged instructions may be executed

and in which protected areas of memory may be accessed.



Multiprogrammed Batch Systems

- CPU is often idle
  - Even with automatic job sequencing.
  - I/O devices are slow compared to processor

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

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

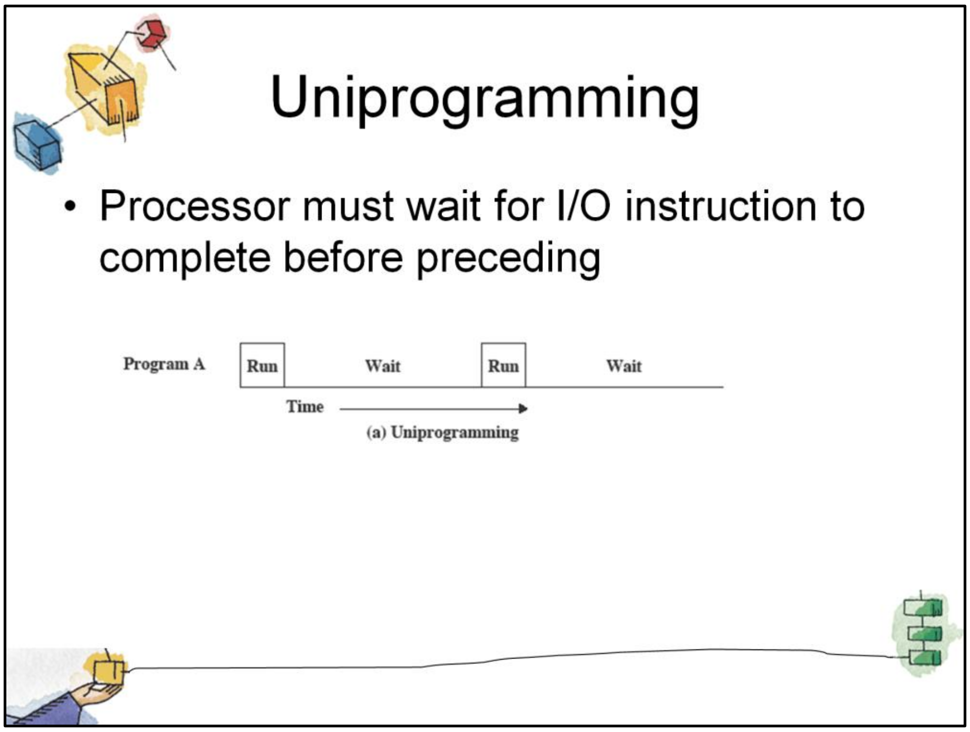
Figure 2.4 System Utilization Example

Even with the automatic job sequencing provided by a simple batch operating system, the processor is often idle.

The problem is that I/O devices are slow compared to the processor.

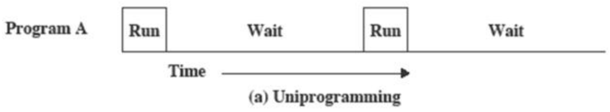
Figure 2.4 details a representative calculation.

- The calculation concerns a program that processes a file of records and performs, on average, 100 machine instructions per record.
- In this example the computer spends over 96% of its time waiting for I/O devices to finish transferring data to and from the file.



Uniprogramming

- Processor must wait for I/O instruction to complete before preceding

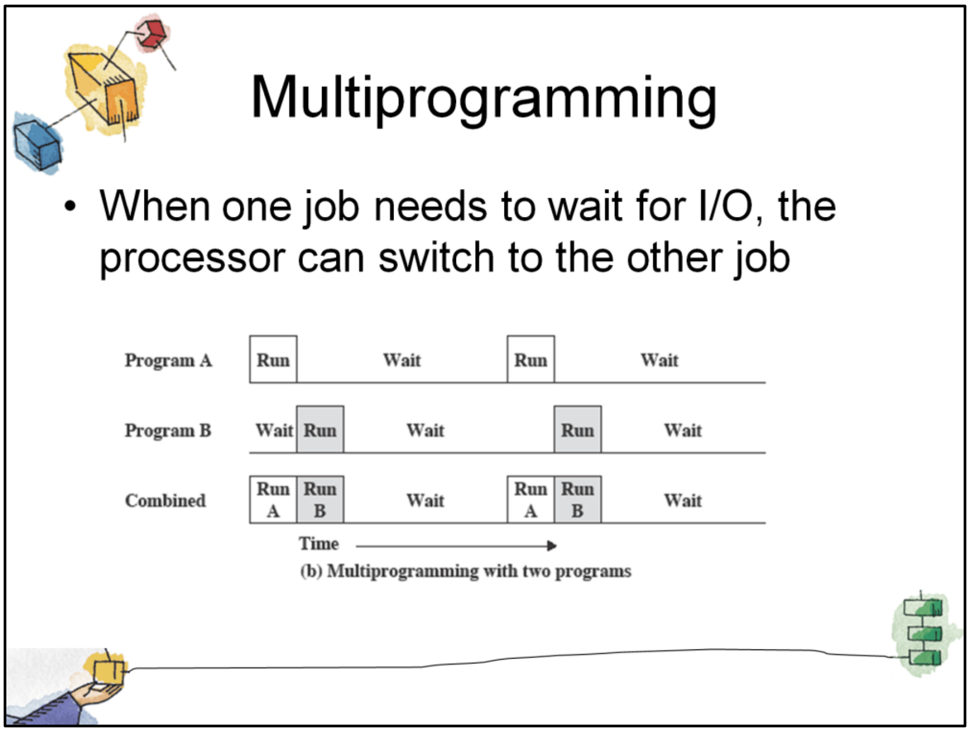


(a) Uniprogramming

In a single application system, uniprogramming, the processor spends a certain amount of time executing, until it reaches an I/O instruction.

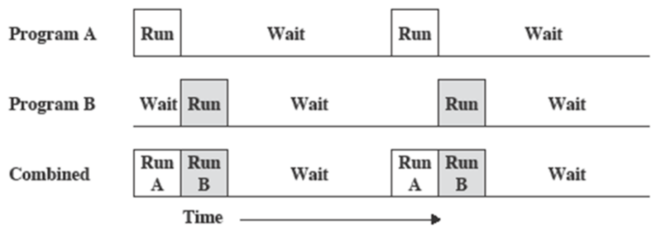
It must then wait until that I/O instruction concludes before proceeding.

This inefficiency is not necessary.



Multiprogramming

- When one job needs to wait for I/O, the processor can switch to the other job

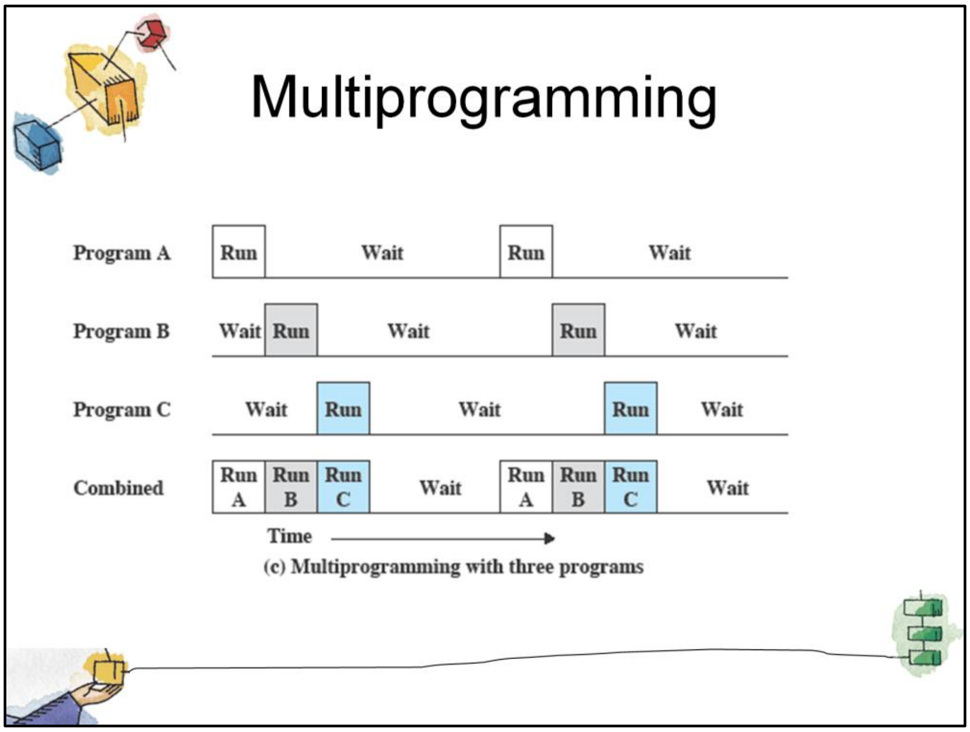


(b) Multiprogramming with two programs

We know that there must be enough memory to hold the OS (resident monitor) and one user program.

Suppose that there is room for the OS and two 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 (Figure 2.5b).



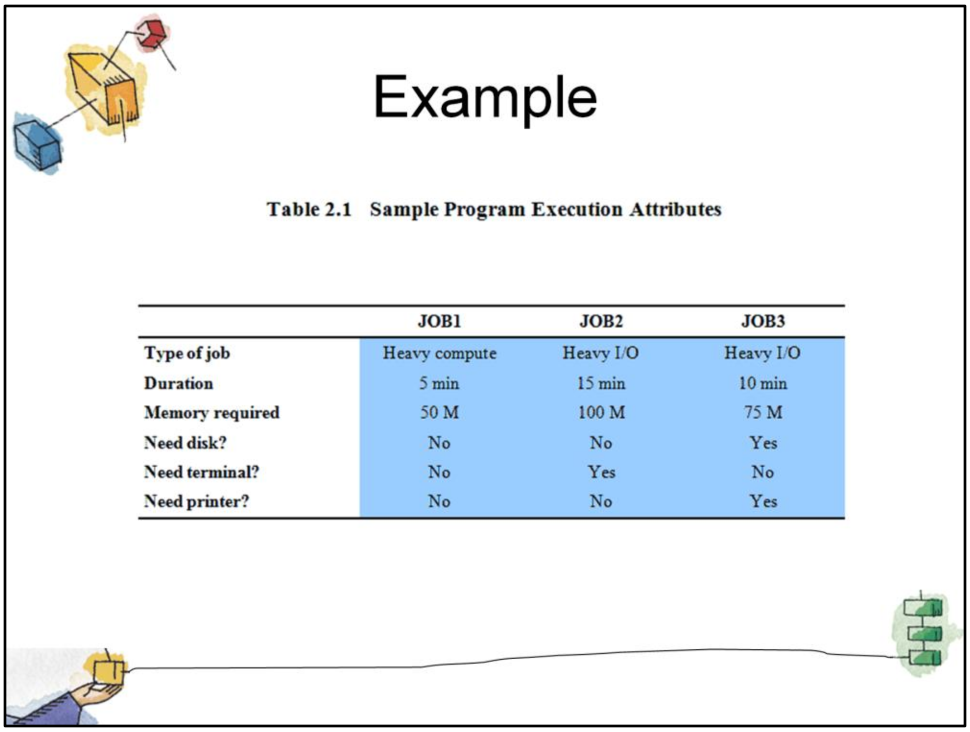
Multiprogramming

(c) Multiprogramming with three programs

Further-more, we might expand memory to hold three, four, or more programs and switch among all of them (Figure 2.5c).

The approach is known as multiprogramming, or multitasking.

It is the central theme of modern operating systems.



Example

Table 2.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

To illustrate the benefit of multiprogramming, we give a simple example.

Consider a computer with 250 Mbytes of available memory (not used by the OS), a disk, a terminal, and a printer.

- Three programs, JOB1, JOB2, and JOB3, are submitted for execution at the same time, with the attributes listed in Table 2.1.

We assume minimal processor requirements for JOB2 and JOB3 and continuous disk and printer use by JOB3. For a simple batch environment, these jobs will be executed in sequence.

- JOB1 completes in 5 minutes.
- JOB2 must wait until the 5 minutes are over and then completes 15 minutes after that.
- JOB3 begins after 20 minutes and completes at 30 minutes from the time it was initially submitted.

The average resource utilization, throughput, and response times are shown in the uniprogramming column of Table 2.2.

Utilization Histograms

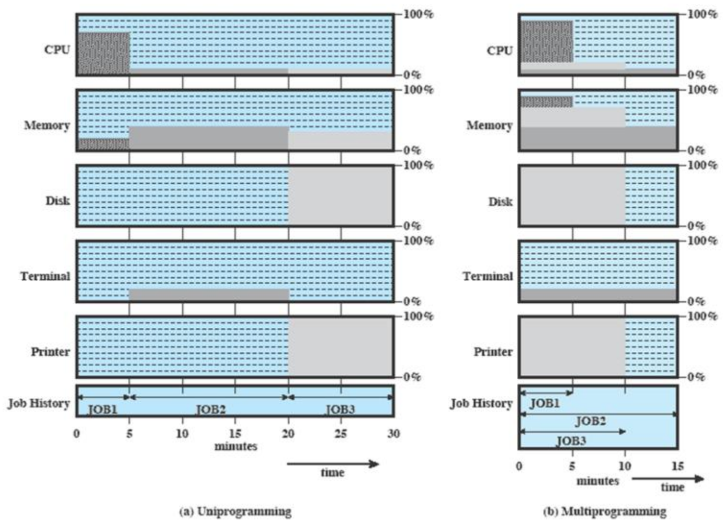


Figure 2.6 Utilization Histograms

Device-by-device utilization of the previous example is illustrated in Figure 2.6a.

It is evident that there is gross underutilization for all resources when averaged over the required 30-minute time period.

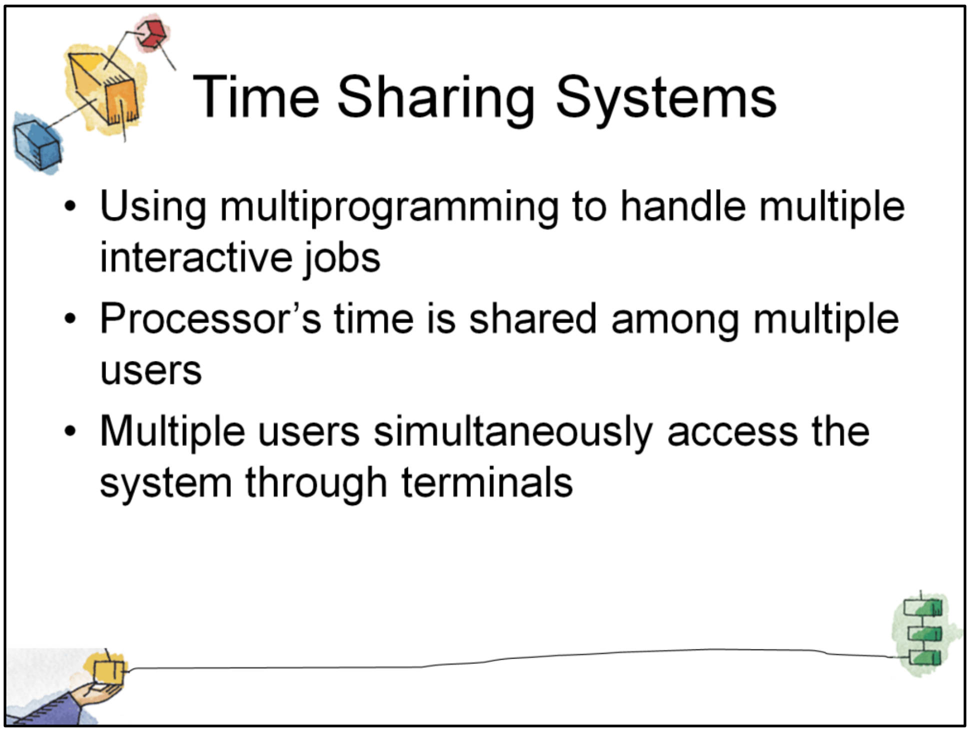
Now suppose that the jobs are run concurrently under a multiprogramming operating system.

- Because there is little resource contention between the jobs, all three can run in nearly minimum time while coexisting with the others in the computer

•JOB1 will still require 5 minutes to complete, but at the end of that time, JOB2 will be one-third finished and JOB3 half finished.

- All three jobs will have finished within 15 minutes.

The improvement is evident when examining the multiprogramming column of Table 2.2, obtained from the histogram shown in Figure 2.6b.



Time Sharing Systems

- Using multiprogramming to handle multiple interactive jobs
- Processor's time is shared among multiple users
- Multiple users simultaneously access the system through terminals

With the use of multiprogramming, batch processing can be quite efficient.

However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer.

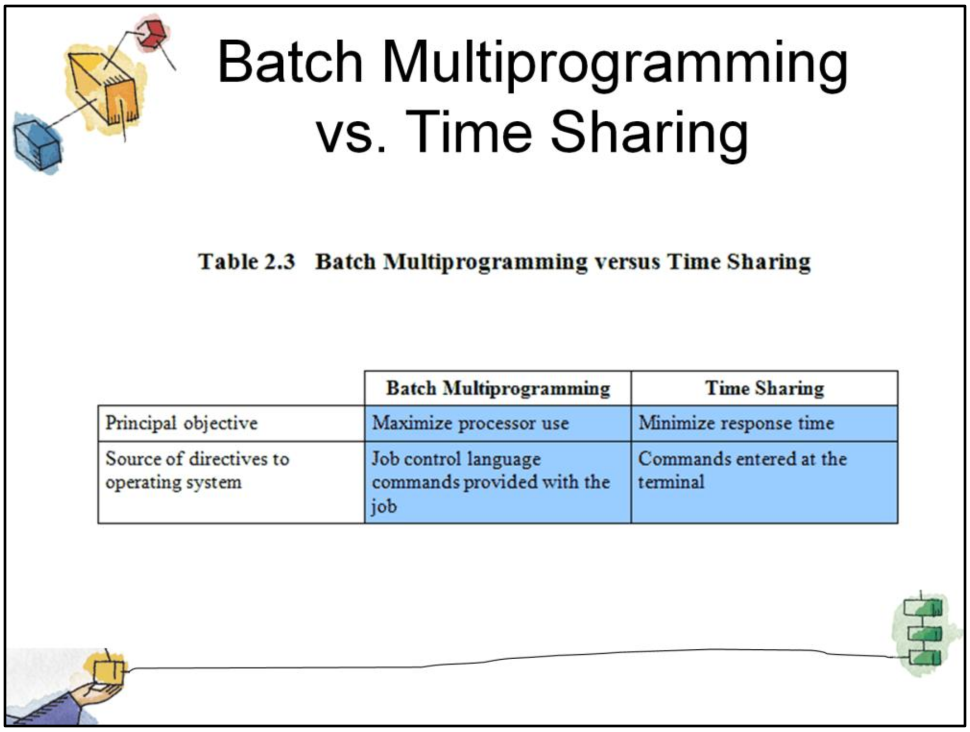
- Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated personal computer or workstation.

- That option was not available in the 1960s, when most computers were big and costly.
- Instead, **time sharing** was developed.

The technique is referred to as time sharing, because processor time is shared among multiple users.

- In a time-sharing system, 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.



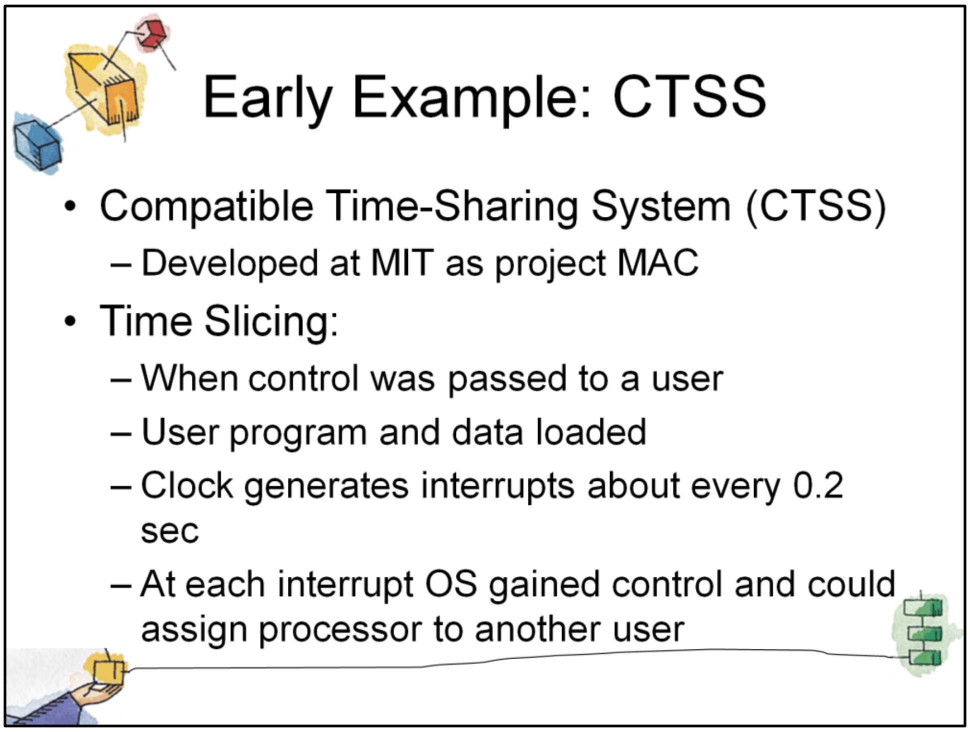
Batch Multiprogramming vs. Time Sharing

Table 2.3 Batch Multiprogramming versus Time Sharing

	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

Both batch processing and time sharing use multiprogramming.

The key differences are listed here



Early Example: CTSS

- Compatible Time-Sharing System (CTSS)
  - Developed at MIT as project MAC
- Time Slicing:
  - When control was passed to a user
  - User program and data loaded
  - Clock generates interrupts about every 0.2 sec
  - At each interrupt OS gained control and could assign processor to another user

One of the first time-sharing operating systems to be developed was the Compatible Time-Sharing System (CTSS)

- developed at MIT by a group known as Project MAC

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 to simplified both the monitor and memory management.
- 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.

At regular time intervals, the current user would be preempted and another user loaded in.

- To preserve the old user program status for later resumption, the old user programs and data were written out to disk before the new user programs and data were read in.
- Subsequently, the old user program code and data were restored in main memory when that program was next given a turn.

CTSS Operation

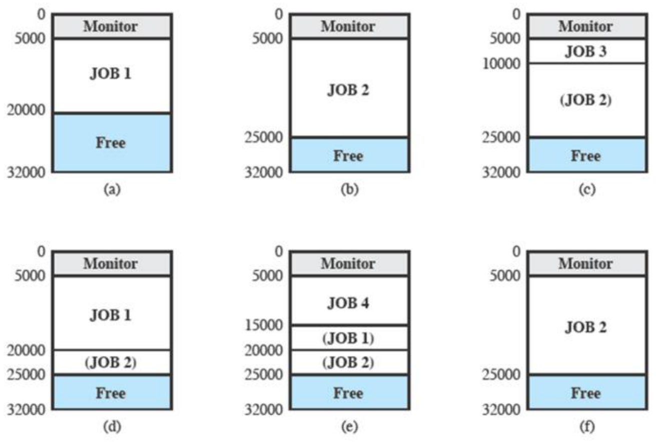


Figure 2.7 CTSS Operation

To minimize disk traffic, user memory was only written out when the incoming program would overwrite it.

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

Initially, the monitor loads JOB1 and transfers control to it (a).

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 (b).

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 (c).

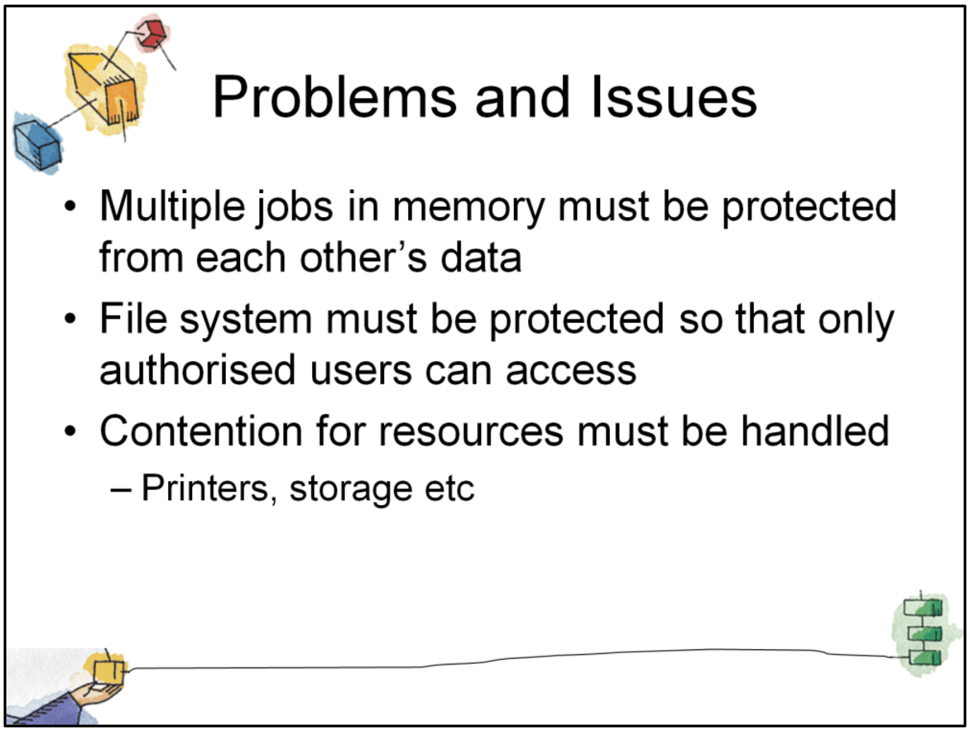
Later, the monitor decides to transfer control back to JOB1.

- An additional portion of JOB2 must be written out when JOB1 is loaded back into memory (d).

When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained (e).

- 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 (f).



Problems and Issues

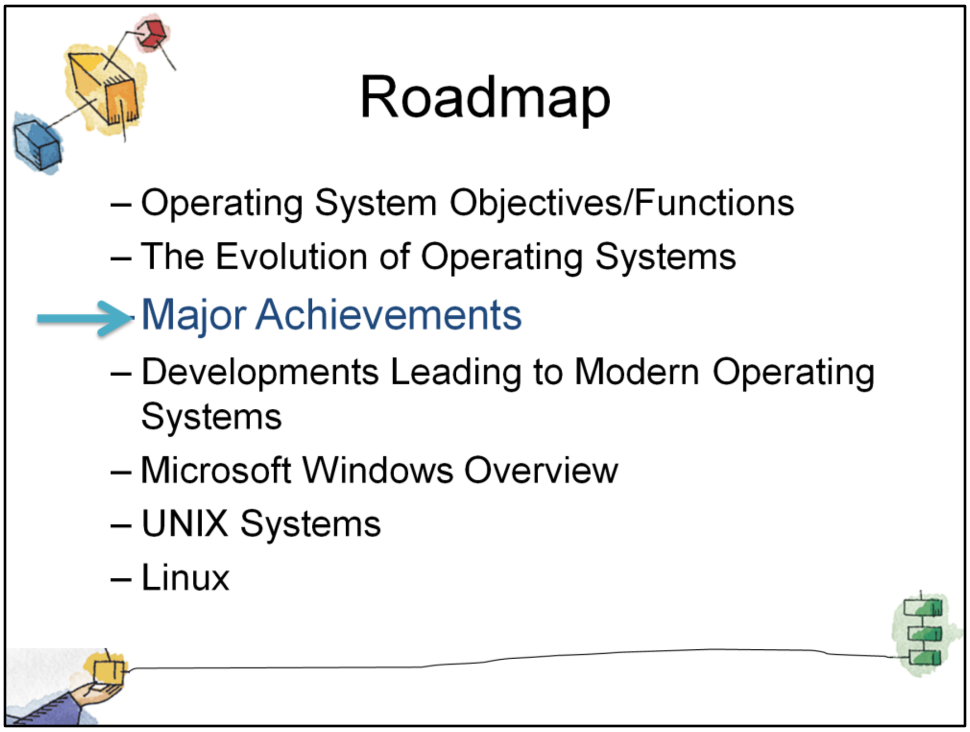
- Multiple jobs in memory must be protected from each other's data
- File system must be protected so that only authorised users can access
- Contention for resources must be handled
  - Printers, storage etc

Time sharing and multiprogramming raise a host of new problems for the OS.

If multiple jobs are in memory, then they must be protected from interfering with each other by, for example, modifying each other's data.

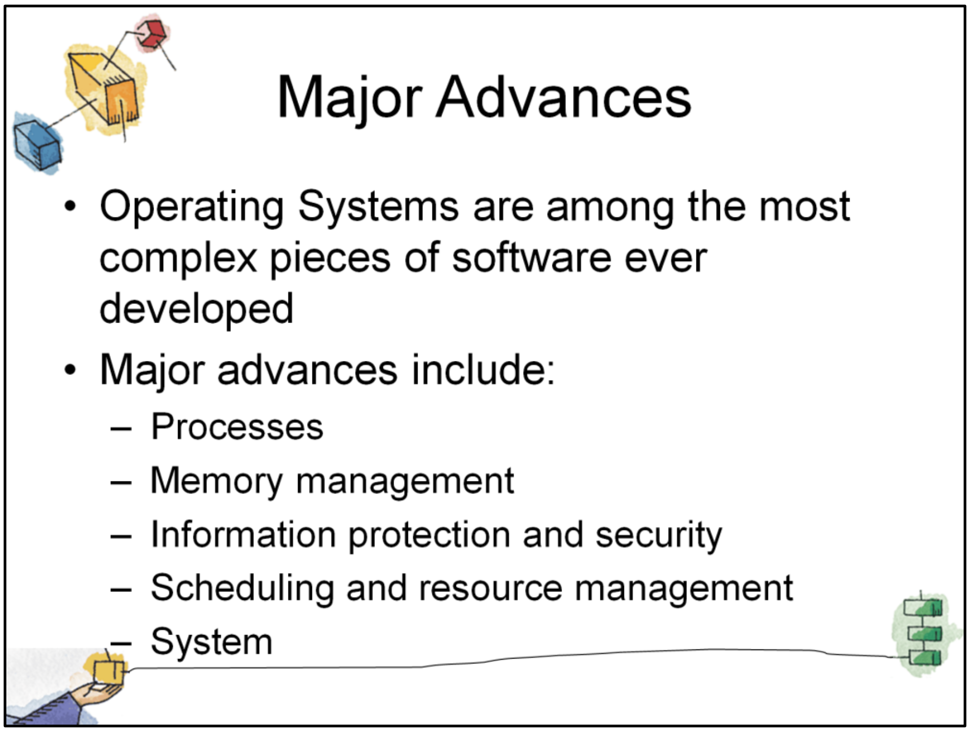
With multiple interactive users, the file system must be protected so that only authorized users have access to a particular file.

The contention for resources, such as printers and mass storage devices, must be handled.



Roadmap

- Operating System Objectives/Functions
- The Evolution of Operating Systems
- → Major Achievements
- Developments Leading to Modern Operating Systems
- Microsoft Windows Overview
- UNIX Systems
- Linux



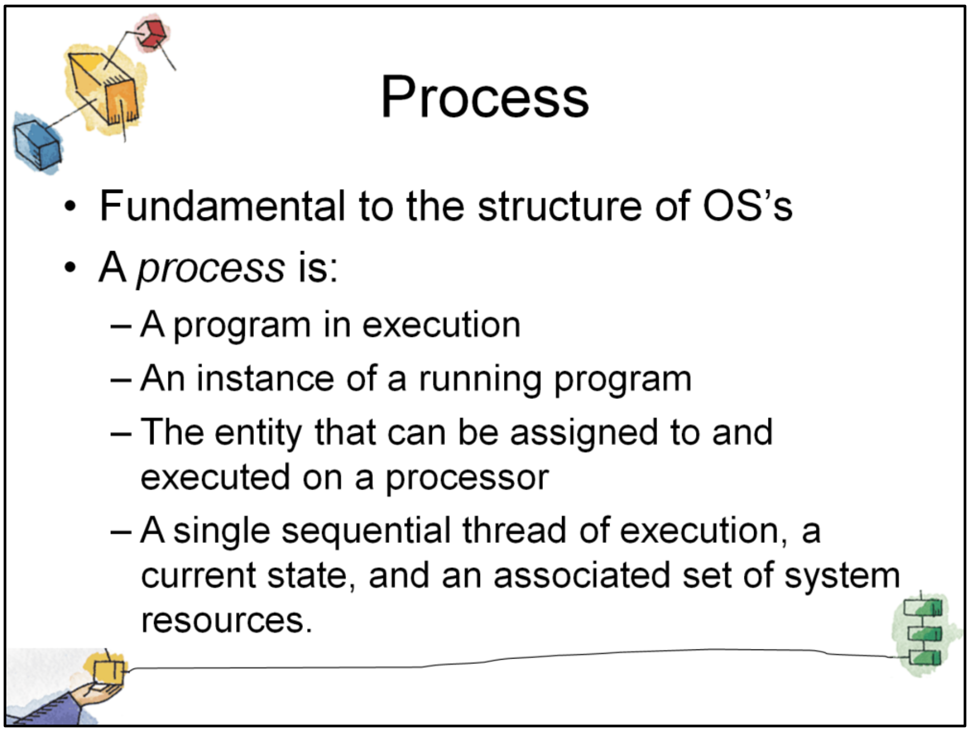
Major Advances

- Operating Systems are among the most complex pieces of software ever developed
- Major advances include:
  - Processes
  - Memory management
  - Information protection and security
  - Scheduling and resource management
  - System

Operating systems are among the most complex pieces of software ever developed.

Five major theoretical advances in the development of operating systems:

- Processes
- Memory management
- Information protection and security
- Scheduling and resource management
- System



Process

- Fundamental to the structure of OS's
- A *process* is:
  - A program in execution
  - An instance of a running program
  - The entity that can be assigned to and executed on a processor
  - A single sequential thread of execution, a current state, and an associated set of system resources.

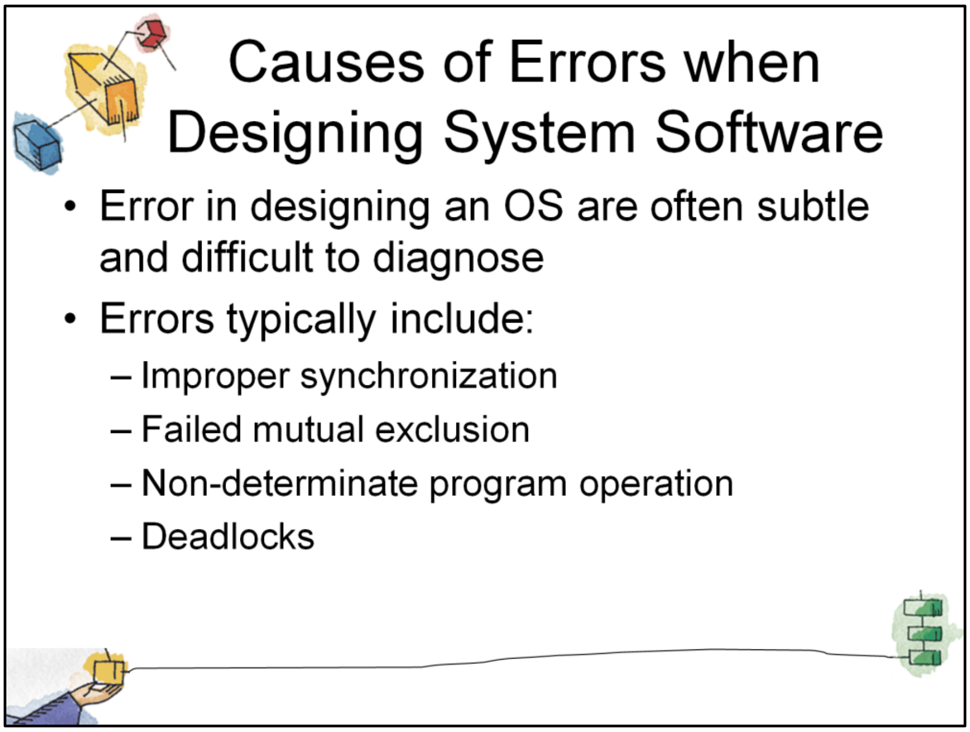
The concept of process is fundamental to the structure of operating systems.

This term was first used by the designers of Multics in the 1960s.

- It is a somewhat more general term than job.

Many definitions have been given for the term process, including

- A program in execution
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources.



Causes of Errors when Designing System Software

- Error in designing an OS are often subtle and difficult to diagnose
- Errors typically include:
  - Improper synchronization
  - Failed mutual exclusion
  - Non-determinate program operation
  - Deadlocks

Efforts to design a system were vulnerable to subtle programming errors whose effects could be observed only when certain relatively rare sequences of actions occurred.

- These errors were difficult to diagnose because they needed to be distinguished from application software errors and hardware errors.
- Even when the error was detected, it was difficult to determine the cause, because the precise conditions under which the errors appeared were very hard to reproduce.

In general terms, there are four main causes of such errors:

Improper synchronization:

- Often a routine must be suspended awaiting an event elsewhere in the system.
  - e.g. a program that initiates an I/O read must wait until the data are available in a buffer before proceeding.
- In such cases, a signal from some other routine is required.
- Improper design of the signalling mechanism can result in signals being lost or duplicate signals being received.

Failed mutual exclusion:

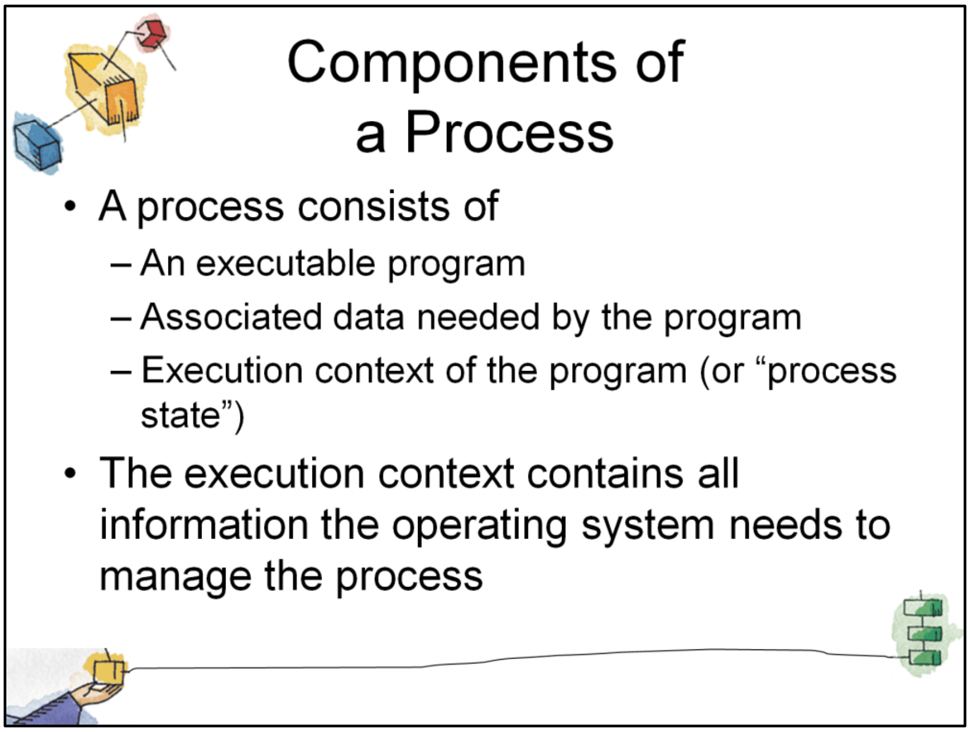
- Often more than one user or program will attempt to make use of a shared resource at the same time.
  - e.g. two users may attempt to edit the same file at the same time.
- If these accesses are not controlled, an error can occur.
- There must be some sort of mutual exclusion mechanism that permits only one routine at a time to perform an update against the file.

Nondeterminate program operation:

- The results of a particular program normally should depend only on the input to that program and not on the activities of other programs in a shared system.
- But when programs share memory, and their execution is interleaved by the processor, they may interfere with each other by overwriting common memory areas in unpredictable ways.
- Thus, the order in which various programs are scheduled may affect the outcome of any particular program.

Deadlocks:

- It is possible for two or more programs to be hung up waiting for each other.
  - e.g. two programs may each require two I/O devices to perform some operation (e.g., disk to tape copy).
  - One of the programs has seized control of one of the devices and the other program has control of the other device.
  - Each is waiting for the other program to release the desired resource.
- Such a deadlock may depend on the chance timing of resource allocation and release.



Components of a Process

- A process consists of
  - An executable program
  - Associated data needed by the program
  - Execution context of the program (or "process state")
- The execution context contains all information the operating system needs to manage the process

We can think of a process as consisting of three components:

- An executable program
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program

This last element is essential.

The execution context, or process state, is the internal data by which the OS is able to supervise and control the process.

- This internal information is separated from the process, because the OS has information not permitted to the process.

The context includes all of the information that the OS needs to manage the process and that the processor needs to execute the process properly including:

- contents of the various processor registers, such as the program counter and data registers
- And information of use to the OS, such as the priority of the process and whether the process is waiting for the completion of a particular I/O event.

Process Management

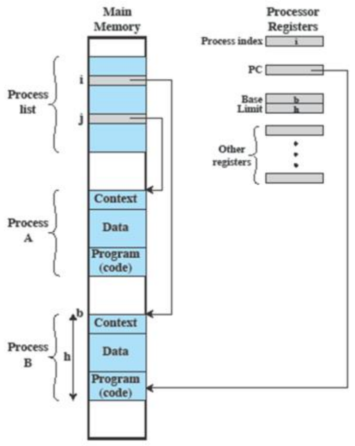


Figure 2.8 Typical Process Implementation

This figure indicates a way in which processes may be managed.

Two processes, A and B, exist in portions of main memory.

- i.e. a block of memory is allocated to each process that contains the program, data, and context information.
- Each process is recorded in a process list built and maintained by the OS.

The process list contains one entry for each process, which includes a pointer to the location of the block of memory that contains the process.

- The entry may also include part or all of the execution context of the process.

The remainder of the execution context is stored elsewhere, perhaps with the process itself (as indicated in this figure) or frequently in a separate region of memory.

The process index register contains the index into the process list of the process currently controlling the processor.

The program counter points to the next instruction in that process to be executed.

The base and limit registers define the region in memory occupied by the process:

- The base register is the starting address of the region of memory and the limit is the size of the region (in bytes or words).
- The program counter and all data references are interpreted relative to the base register and must not exceed the value in the limit register.
- This prevents interprocess interference.

In Figure 2.8, the process index register indicates that process B is executing.

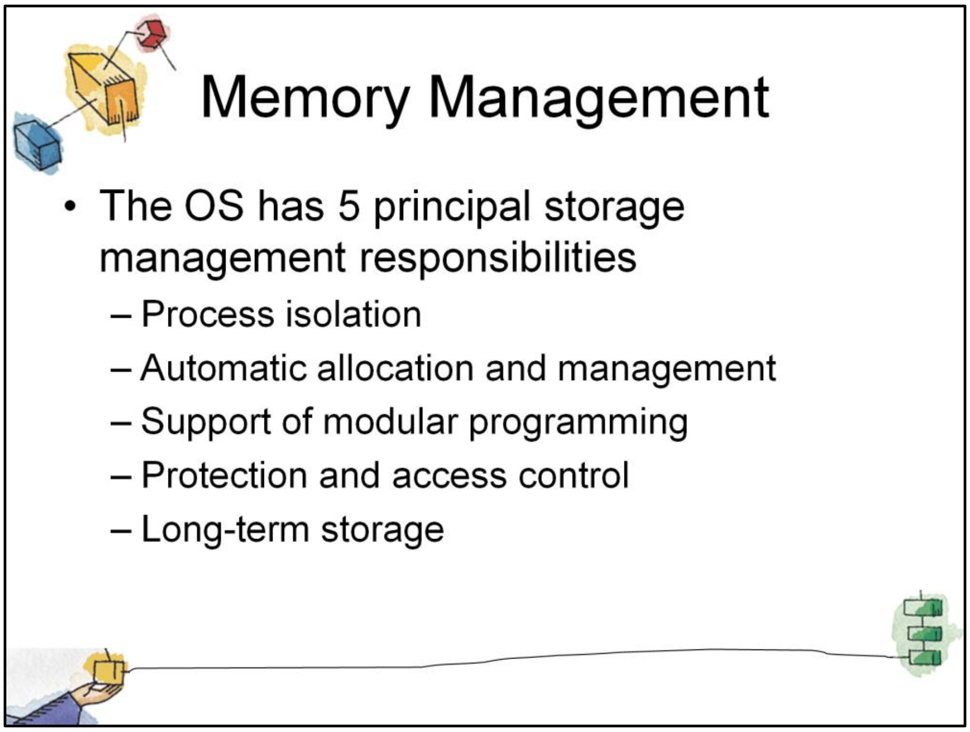
- Process A was previously executing but has been temporarily interrupted.

The contents of all the registers at the moment of A’s interruption were recorded in its execution context.

Later, the OS can perform a process switch and resume execution of process A.

The process switch consists of storing the context of B and restoring the context of A.

When the program counter is loaded with a value pointing into A’s program area, process A will automatically resume execution.



Memory Management

- The OS has 5 principal storage management responsibilities
  - Process isolation
  - Automatic allocation and management
  - Support of modular programming
  - Protection and access control
  - Long-term storage

The needs of users can be met best by a computing environment that supports modular programming and the flexible use of data.

System managers need efficient and orderly control of storage allocation.

The OS, to satisfy these requirements, has five principal storage management responsibilities:

Process isolation:

- The OS must prevent independent processes from interfering with each other's memory, both data and instructions.

Automatic allocation and management:

- Programs should be dynamically allocated across the memory hierarchy as required.
- Allocation should be transparent to the programmer.
- Thus, the programmer is relieved of concerns relating to memory limitations, and the OS can achieve efficiency by assigning memory to jobs only as needed.

Support of modular programming:

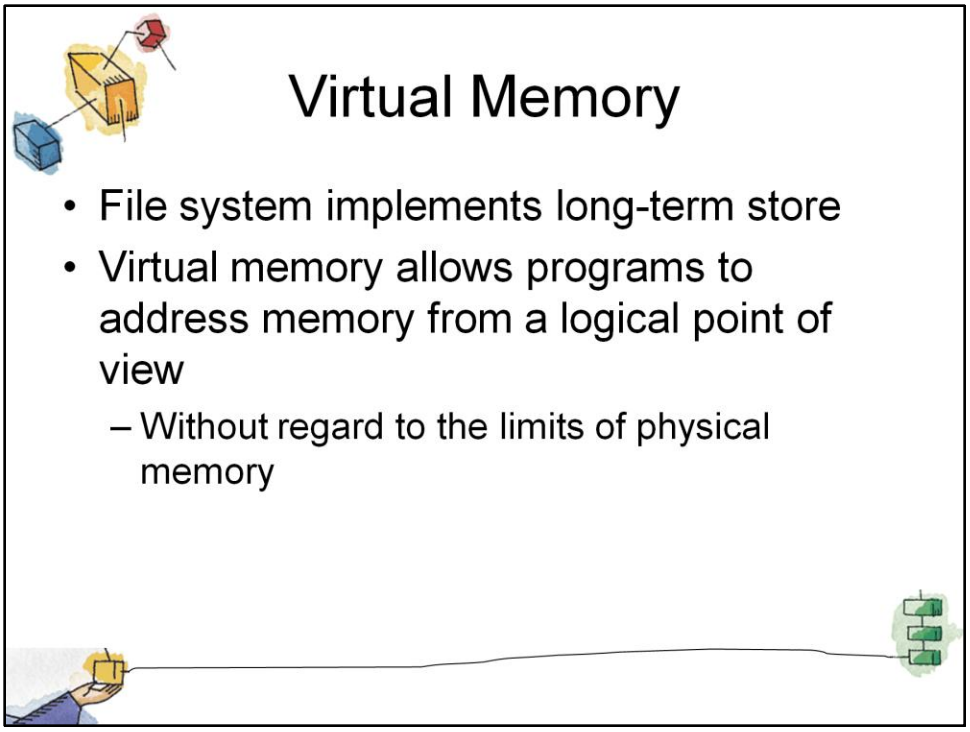
- Programmers should be able to define program modules, and to create, destroy, and alter the size of modules dynamically.

Protection and access control:

- Sharing of memory, at any level of the memory hierarchy, creates the potential for one program to address the memory space of another.
- This is desirable when sharing is needed by particular applications.
- At other times, it threatens the integrity of programs and even of the OS itself.
- The OS must allow portions of memory to be accessible in various ways by various users.

Long-term storage:

- Many application programs require means for storing information for extended periods of time, after the computer has been powered down.



Virtual Memory

- File system implements long-term store
- Virtual memory allows programs to address memory from a logical point of view
  - Without regard to the limits of physical memory

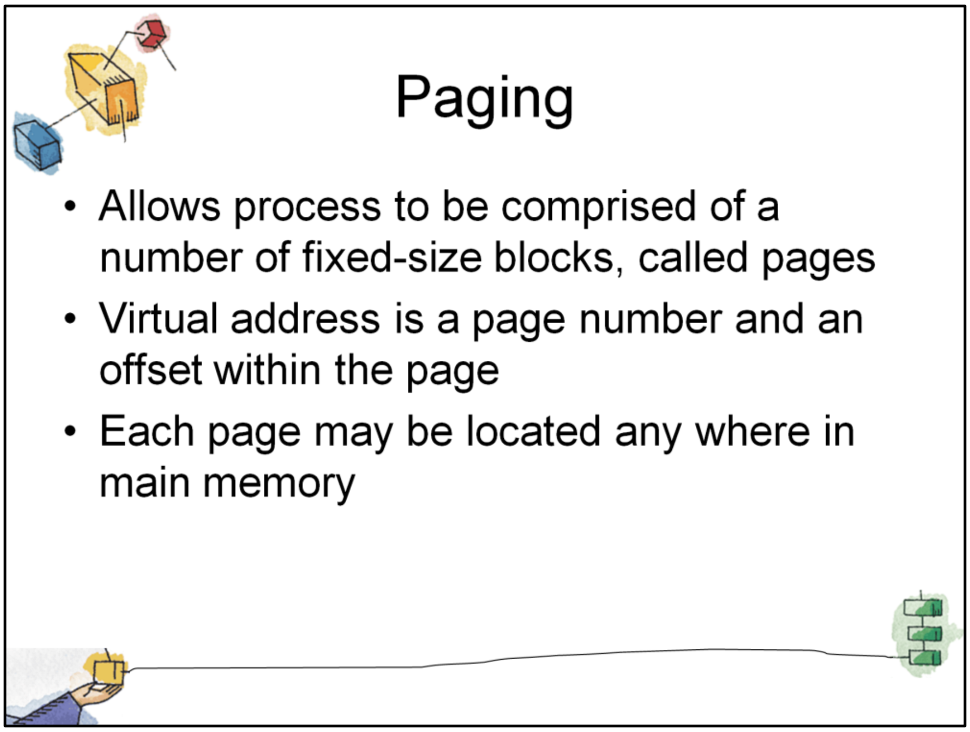
Typically, operating systems meet the preceding requirements with virtual memory and file system facilities.

The file system implements a long-term store,

• with information stored in named objects, called files.

• The file is a convenient concept for the programmer and is a useful unit of access control and protection for the OS.

Virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available.



Paging

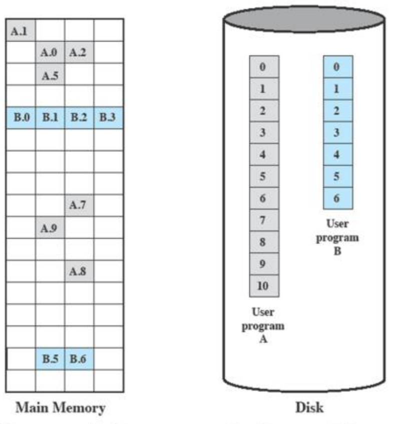
- Allows process to be comprised of a number of fixed-size blocks, called pages
- Virtual address is a page number and an offset within the page
- Each page may be located any where in main memory

Paging systems were introduced, which allow processes to be comprised of a number of fixed-size blocks, called pages.

A program references a word by means of a virtual address consisting of a page number and an offset within the page.

- Each page of a process may be located anywhere in main memory.

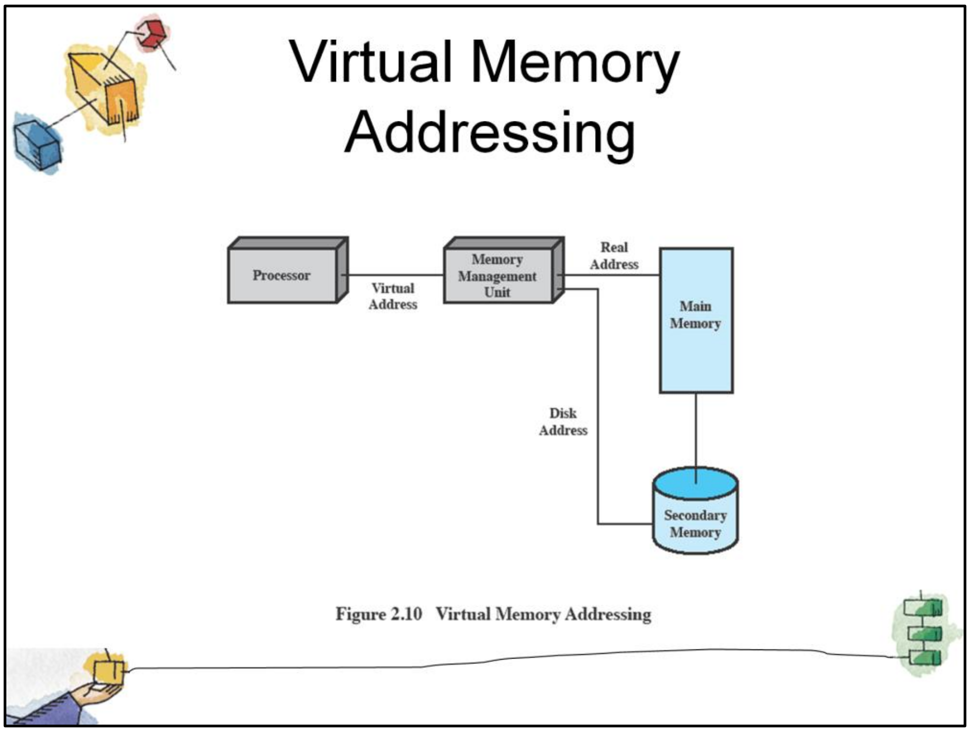
Virtual Memory



Main memory consists of a number of fixed-length frames, each equal to the size of a page. For a program to execute, some or all of its pages must be in main memory.

Secondary memory (disk) can hold many fixed-length pages. A user program consists of some number of pages. Pages for all programs plus the operating system are on disk, as are files.

Figure 2.9 Virtual Memory Concepts



Virtual Memory Addressing

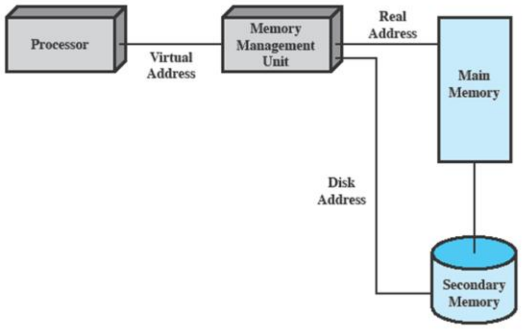


Figure 2.10 Virtual Memory Addressing

The paging system provides for a dynamic mapping between the virtual address used in the program and a real address, or physical address, in main memory.

This figure highlights the addressing concerns in a virtual memory scheme.

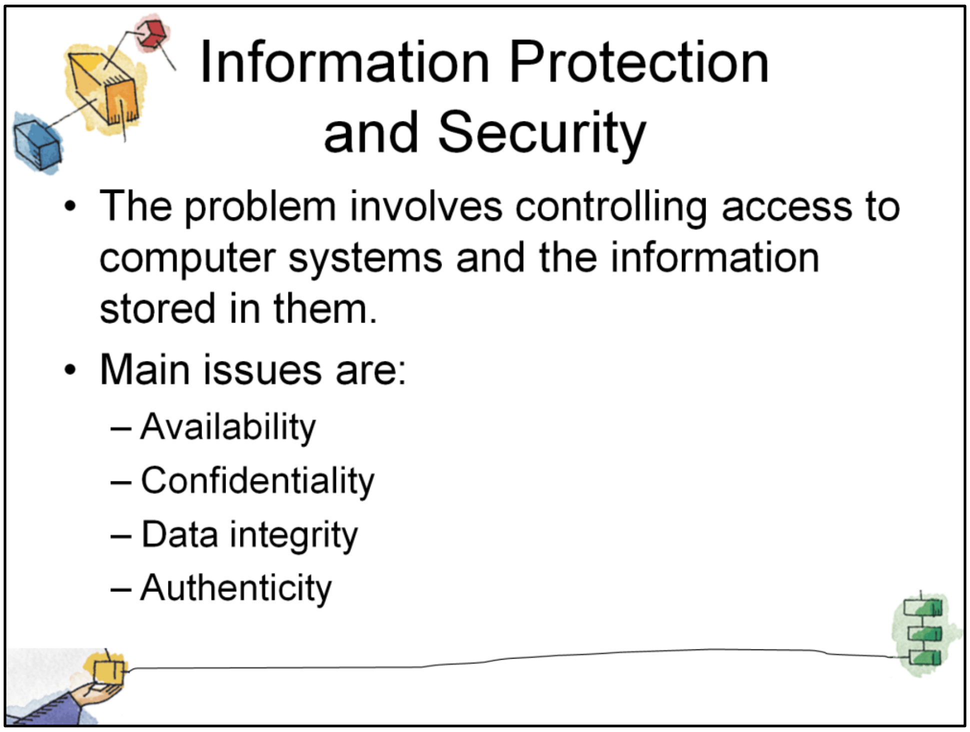
Storage consists of directly addressable (by machine instructions) main memory and lower-speed auxiliary memory that is accessed indirectly by loading blocks into main memory.

Address translation hardware (memory management unit) is interposed between the processor and memory.

- Programs reference locations using virtual addresses, which are mapped into real main memory addresses.
- If a reference is made to a virtual address not in real memory, then a portion of the contents of real memory is swapped out to auxiliary memory and the desired block of data is swapped in.
- During this activity, the process that generated the address reference must be suspended.

The OS designer needs to develop an address translation mechanism that generates little overhead and a storage allocation policy that minimizes

the traffic between memory levels.



Information Protection and Security

- The problem involves controlling access to computer systems and the information stored in them.
- Main issues are:
  - Availability
  - Confidentiality
  - Data integrity
  - Authenticity

The growth in the use of time-sharing systems and, more recently, computer networks has brought with it a growth in concern for the protection of information.

There are some general-purpose tools that can be built into computers and operating systems that support a variety of protection and security mechanisms.

- In general, we are concerned with the problem of controlling access to computer systems and the information stored in them.

Much of the work in security and protection as it relates to operating systems can be roughly grouped into four categories:

Availability:

- Concerned with protecting the system against interruption

Confidentiality:

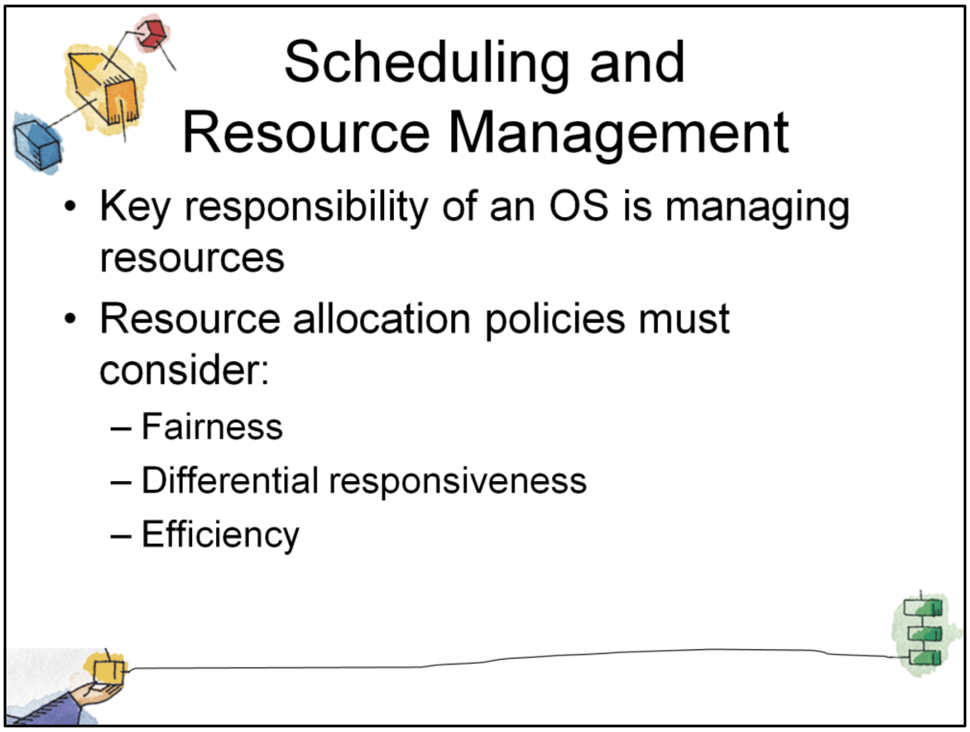
- Assures that users cannot read data for which access is unauthorized

Data integrity:

- Protection of data from unauthorized modification

Authenticity:

- Concerned with the proper verification of the identity of users and the validity of messages or data



Scheduling and Resource Management

- Key responsibility of an OS is managing resources
- Resource allocation policies must consider:
  – Fairness
  – Differential responsiveness
  – Efficiency

A key responsibility of the OS is to manage the various resources available to it (main memory space, I/O devices, processors) and to schedule their use by the various active processes.

Any resource allocation and scheduling policy must consider three factors:

Fairness:

- Typically, we would like all processes that are competing for the use of a particular resource to be given approximately equal and fair access to that resource.
- This is especially so for jobs of the same class, that is, jobs of similar demands.

Differential responsiveness:

- On the other hand, the OS may need to discriminate among different classes of jobs with different service requirements.
- The OS should attempt to make allocation and scheduling decisions to meet the total set of requirements.
- The OS should also make these decisions dynamically.
- e.g. if a process is waiting for the use of an I/O device, the OS may wish to schedule that process for execution as soon as possible to free up the device for later demands from other processes.

Efficiency:

- The OS should attempt to
  - maximize throughput,
  - minimize response time, and
  - accommodate as many users as possible.
- These criteria conflict;
- Finding the right balance for a particular situation is an ongoing problem for operating system research.

Key Elements of an Operating System

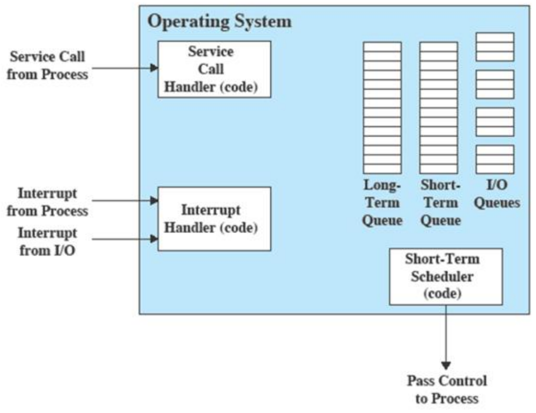


Figure 2.11 Key Elements of an Operating System for Multiprogramming

This figure suggests the major elements of the OS involved in the scheduling of processes and the allocation of resources in a multiprogramming environment.

The OS maintains a number of queues, each of which is simply a list of processes waiting for some resource.

The short-term queue consists of processes that are in main memory (or at least an essential minimum portion of each is in main memory) and are ready to run as soon as the processor is made available.

- Any one of these processes could use the processor next.

It is up to the short-term scheduler, or dispatcher, to pick one.

- A common strategy is to give each process in the queue some time in turn;
- this is referred to as **a round-robin technique**.
- Another strategy is to assign priority levels to the various processes, with the scheduler selecting processes in priority order.

The long-term queue is a list of new jobs waiting to use the processor.

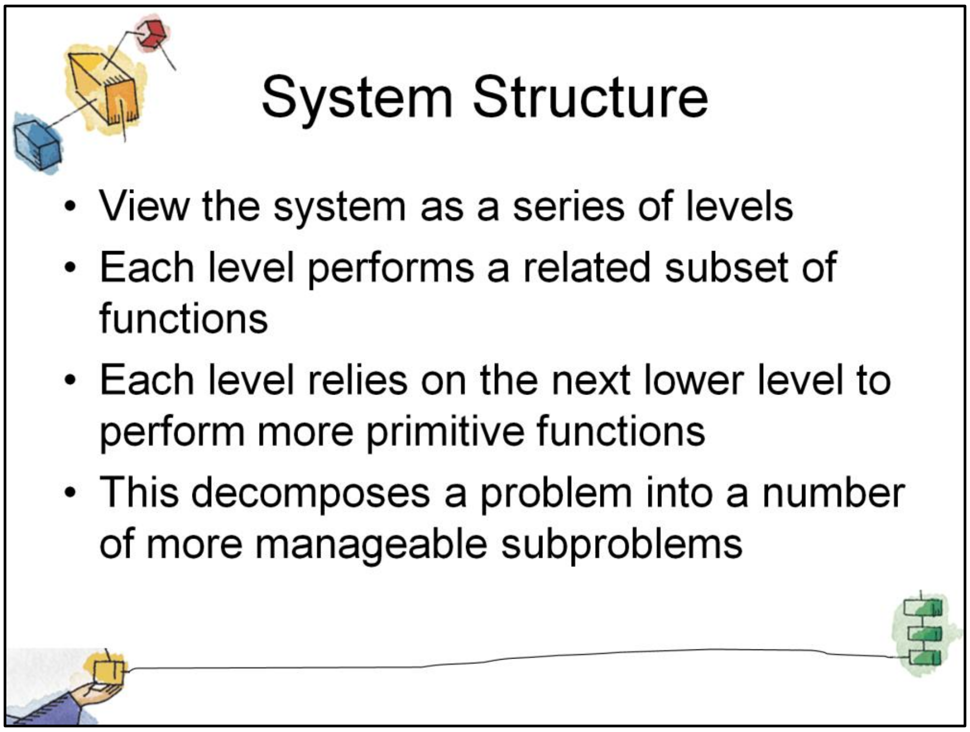
- The OS adds jobs to the system by transferring a process from the long-term queue to the short-term queue.
- At that time, a portion of main memory must be allocated to the incoming process.
- Thus, the OS must be sure that it does not overcommit memory or processing time by admitting too many processes to the system.

There is an I/O queue for each I/O device.

- More than one process may request the use of the same I/O device.
- All processes waiting to use each device are lined up in that device's queue.
- Again, the OS must determine which process to assign to an available I/O device.

The OS receives control of the processor at the interrupt handler if an interrupt occurs.

- A process may specifically invoke some operating system service, such as an I/O device handler by means of a service call.
- In this case, a service call handler is the entry point into the OS. In any case, once the interrupt or service call is handled, the short-term scheduler is invoked to pick a process for execution.



System Structure

- View the system as a series of levels
- Each level performs a related subset of functions
- Each level relies on the next lower level to perform more primitive functions
- This decomposes a problem into a number of more manageable subproblems

There has been increasing use of the concepts of hierarchical layers and information abstraction.

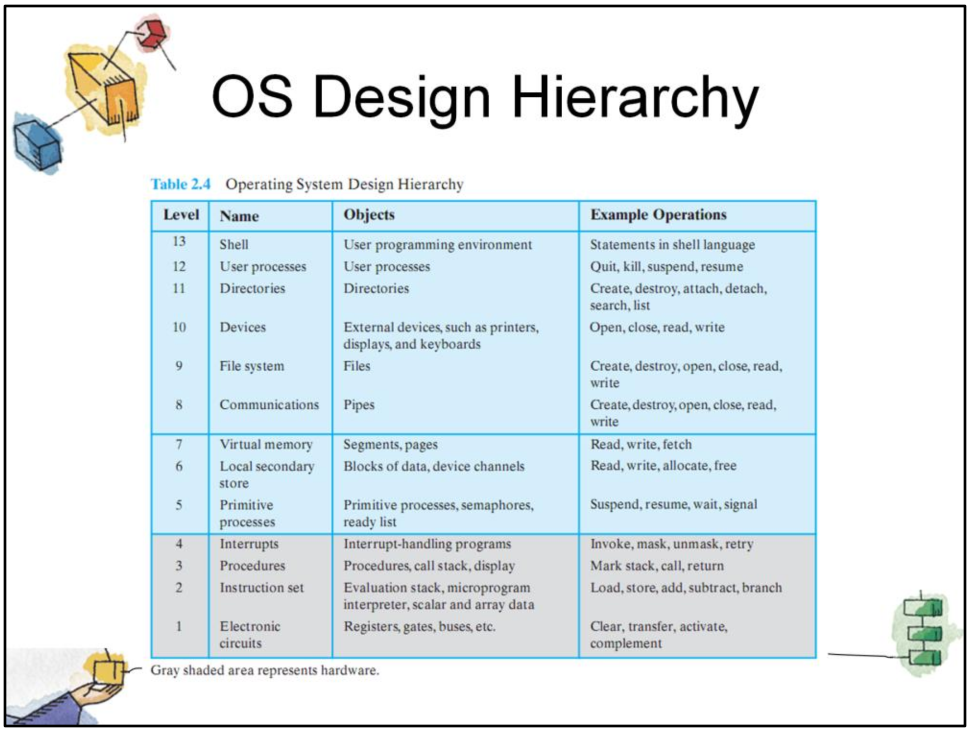
The hierarchical structure of a modern OS separates its functions according to their characteristic time scale and their level of abstraction.

We can view the system as a series of levels.

- Each level performs a related subset of the functions required of the OS.
- It relies on the next lower level to perform more primitive functions and to conceal the details of those functions.
- It provides services to the next higher layer.

Ideally, the levels should be defined so that changes in one level do not require changes in other levels.

Thus, we have decomposed one problem into a number of more manageable subproblems.



OS Design Hierarchy

Table 2.4 Operating System Design Hierarchy

Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printers, displays, and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Virtual memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive processes, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalar and array data	Load, store, add, subtract, branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

Gray shaded area represents hardware.

NON OS Levels

Level 1: Consists of electronic circuits,

• the objects that are dealt with are registers, memory cells, and logic gates.

• The operations defined on these objects are actions, such as clearing a register or reading a memory location.

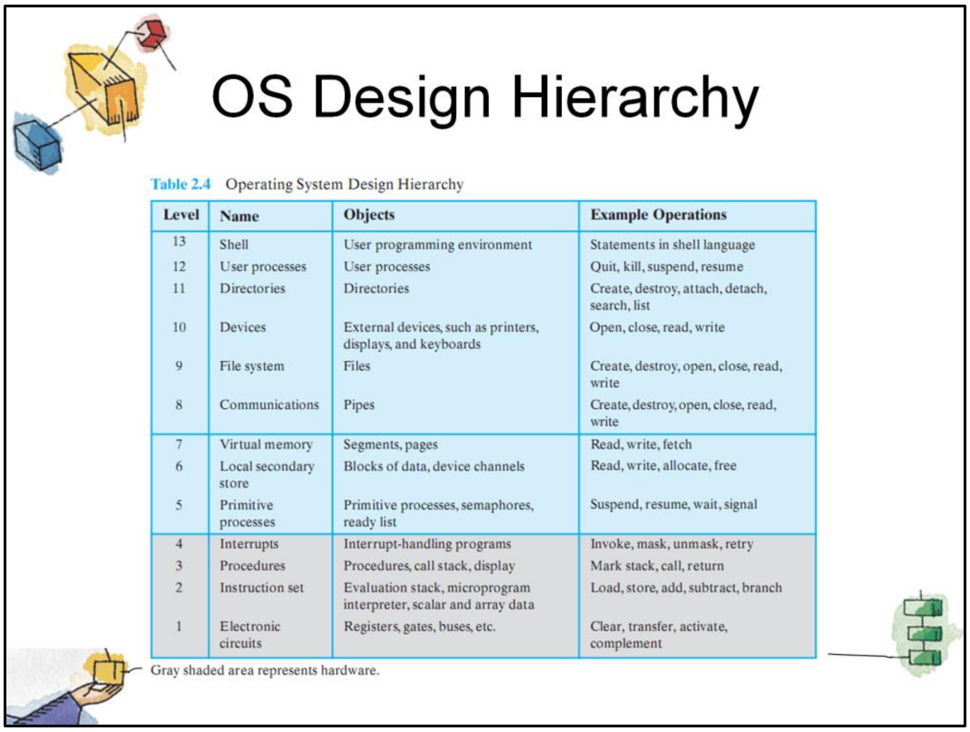
Level 2: The processor’s instruction set.

• The operations at this level are those allowed in the machine language instruction set, such as add, subtract, load, and store.

Level 3: Adds the concept of a procedure or subroutine, plus the call/return operations.

Level 4: Introduces interrupts, which cause the processor to save the current context and invoke an interrupt-handling routine.

Next picture...



OS Design Hierarchy

Table 2.4 Operating System Design Hierarchy

Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printers, displays, and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Virtual memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive processes, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalar and array data	Load, store, add, subtract, branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

Gray shaded area represents hardware.

LEVELS INVOLVING A SINGLE PROCESSOR

Level 5: The notion of a process as a program in execution is introduced at this level.

- The fundamental requirements on the OS to support multiple processes include the ability to suspend and resume processes.
- Requires saving hardware registers so that execution can be switched from one process to another.
- In addition, if processes need to cooperate, then some method of synchronization is needed.

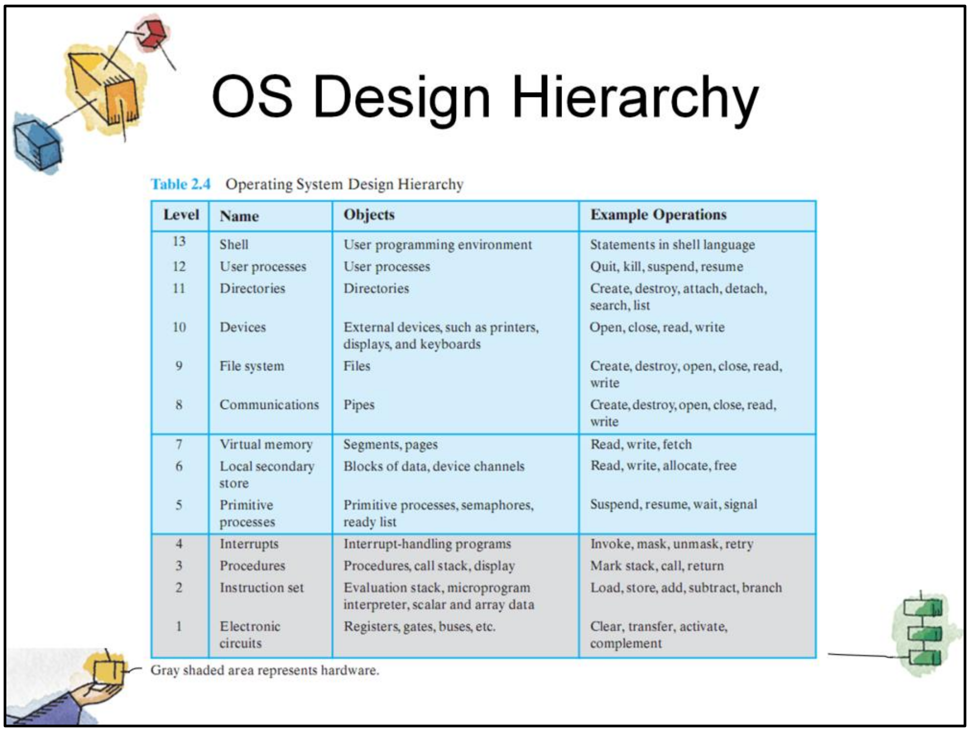
Level 6: Deals with the secondary storage devices of the computer.

- At this level, the functions of positioning the read/write heads and the actual transfer of blocks of data occur.
- Level 6 relies on level 5 to schedule the operation and to notify the requesting process of completion of an operation.
- Higher levels are concerned with the address of the needed data on the disk and provide a request for the appropriate block to a device driver at level 5.

Level 7: Creates a logical address space for processes.

- This level organizes the virtual address space into blocks that can be moved between main memory and secondary memory.
- Three schemes are in common use:
    - using fixed-size pages,
    - using variable-length segments, and
    - using both.
- When a needed block is not in main memory, logic at this level requests a transfer from level 6.

Next picture...



OS Design Hierarchy

Table 2.4 Operating System Design Hierarchy

Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printers, displays, and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Virtual memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive processes, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalar and array data	Load, store, add, subtract, branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

Gray shaded area represents hardware.

MULTIPROCESSOR LEVELS

Level 8: Deals with the communication of information and messages between processes.

- This level deals with a richer sharing of information.
- One of the most powerful tools for this purpose is the pipe, which is a logical channel for the flow of data between processes.
- A pipe is defined with its output from one process and its input into another process.
- It can also be used to link external devices or files to processes.

Level 9: Supports the long-term storage of named files.

- At this level, the data on secondary storage are viewed in terms of abstract, variable-length entities.

Level 10: Provides access to external devices using standardized interfaces.

Level 11: Is responsible for maintaining the association between the external and internal identifiers of the system's resources and objects.

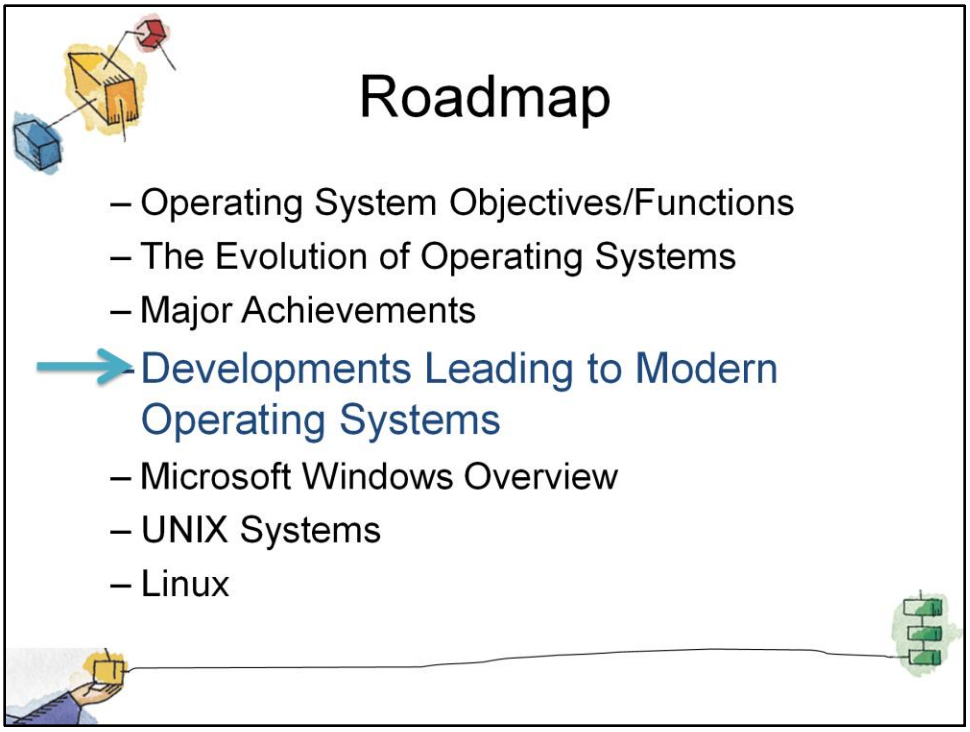
- The external identifier is a name that can be employed by an application or user.
- The internal identifier is an address or other indicator that can be used by lower levels of the OS to locate and control an object.
- These associations are maintained in a directory.
- Entries include not only external/internal mapping, but also characteristics such as access rights.

Level 12: Provides a full-featured facility for the support of processes.

- This goes far beyond what is provided at level 5.At level 5, only the processor register contents associated with a process are maintained, plus the logic for dispatching processes.
- At level 12, all of the information needed for the orderly management of processes is supported.
- This includes the virtual address space of the process, a list of objects and processes with which it may interact and the constraints of that interaction, parameters passed to the process upon creation, and any other characteristics of the process that might be used by the OS to control the process.

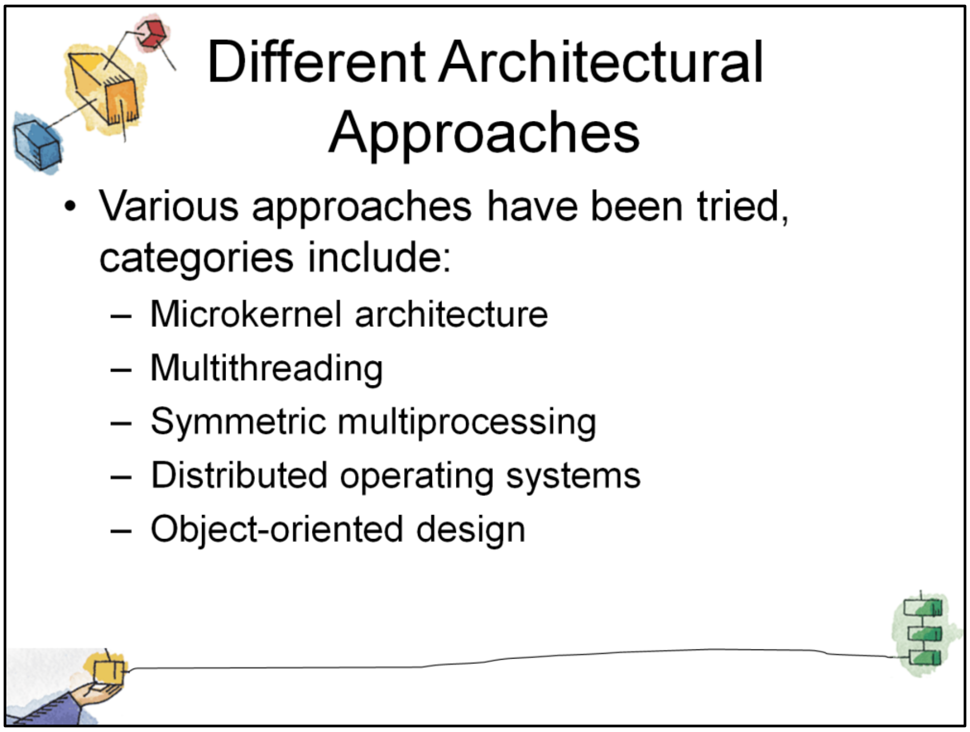
Level 13: Provides an interface to the OS for the user.

- It is referred to as the shell because it separates the user from OS details and presents the OS simply as a collection of services.
- The shell accepts user commands or job control statements, interprets these, and creates and controls processes as needed.
- e.g. the interface at this level could be implemented in a graphical manner, providing the user with commands through a list presented as a menu and displaying results using graphical output to a specific device such as a screen.



Roadmap

- – Operating System Objectives/Functions
- – The Evolution of Operating Systems
- – Major Achievements
- – Developments Leading to Modern Operating Systems
- – Microsoft Windows Overview
- – UNIX Systems
- – Linux



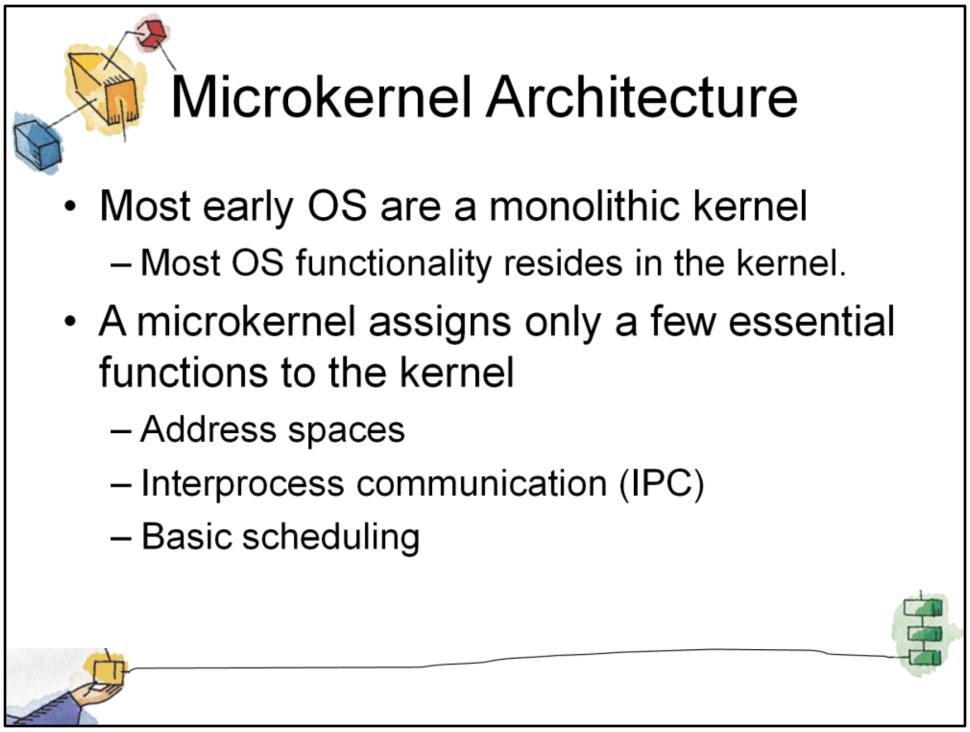
Different Architectural Approaches

- Various approaches have been tried, categories include:
  - Microkernel architecture
  - Multithreading
  - Symmetric multiprocessing
  - Distributed operating systems
  - Object-oriented design

The rate of change in the demands on operating systems requires not just modifications and enhancements to existing architectures but new ways of organizing the OS.

A wide range of different approaches and design elements has been tried in both experimental and commercial operating systems, but much of the work fits into the following categories:

- Microkernel architecture
- Multithreading
- Symmetric multiprocessing
- Distributed operating systems
- Object-oriented design



Microkernel Architecture

- Most early OS are a monolithic kernel
  - Most OS functionality resides in the kernel.
- A microkernel assigns only a few essential functions to the kernel
  - Address spaces
  - Interprocess communication (IPC)
  - Basic scheduling

Most operating systems, until recently, featured a large monolithic kernel.

- Most of what is thought of as OS functionality is provided in these large kernels, including
  - scheduling,
  - file system,
  - networking,
  - device drivers,
  - memory management,
  - and more.
- Typically, a monolithic kernel is implemented as a single process, with all elements sharing the same address space.

A microkernel architecture assigns only a few essential functions to the kernel, including

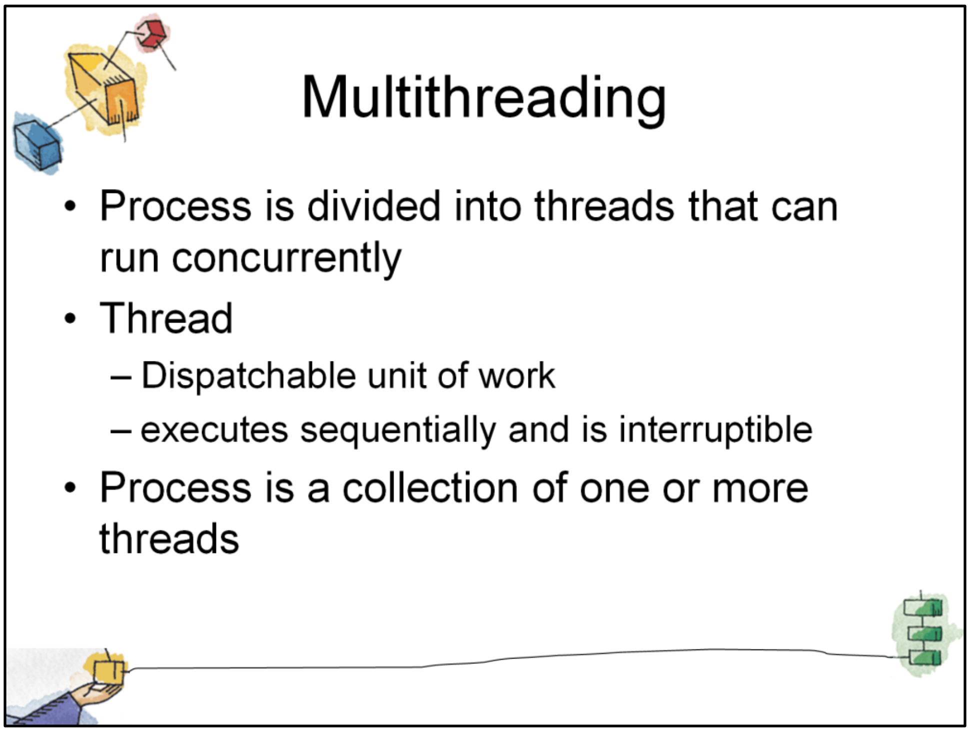
- address spaces,
- interprocess communication (IPC), and
- basic scheduling.

Other OS services are provided by processes, sometimes called servers, that run in user mode and are treated like any other application by the microkernel.

- This approach decouples kernel and server development.
- Servers may be customized to specific application or environment requirements.

The microkernel approach simplifies implementation, provides flexibility, and is well suited to a distributed environment.

- In essence, a microkernel interacts with local and remote server processes in the same way, facilitating construction of distributed systems.



Multithreading

- Process is divided into threads that can run concurrently
- Thread
  - Dispatchable unit of work
  - executes sequentially and is interruptible
- Process is a collection of one or more threads

Multithreading is a technique in which a process, executing an application, is divided into threads that can run concurrently.

We can make the following distinction:

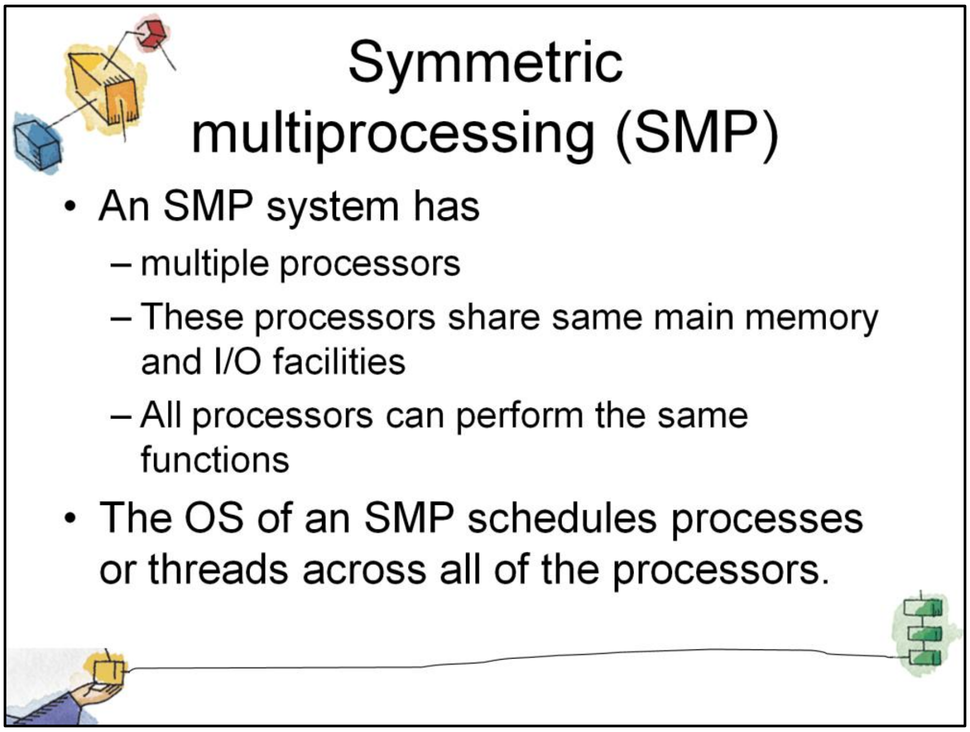
Thread: A dispatchable unit of work.

- It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching).
- A thread executes sequentially and is interruptible so that the processor can turn to another thread.

Process: A collection of one or more threads and associated system resources (such as memory containing both code and data, open files, and devices).

- This corresponds closely to the concept of a program in execution.
- By breaking a single application into multiple threads, the programmer has great control over the modularity of the application and the timing of application-related events.

Multithreading is useful for applications that perform a number of essentially independent tasks that do not need to be serialized.



Symmetric multiprocessing (SMP)

- An SMP system has
  - multiple processors
  - These processors share same main memory and I/O facilities
  - All processors can perform the same functions
- The OS of an SMP schedules processes or threads across all of the processors.

Vendors have now introduced computers with multiple microprocessors.

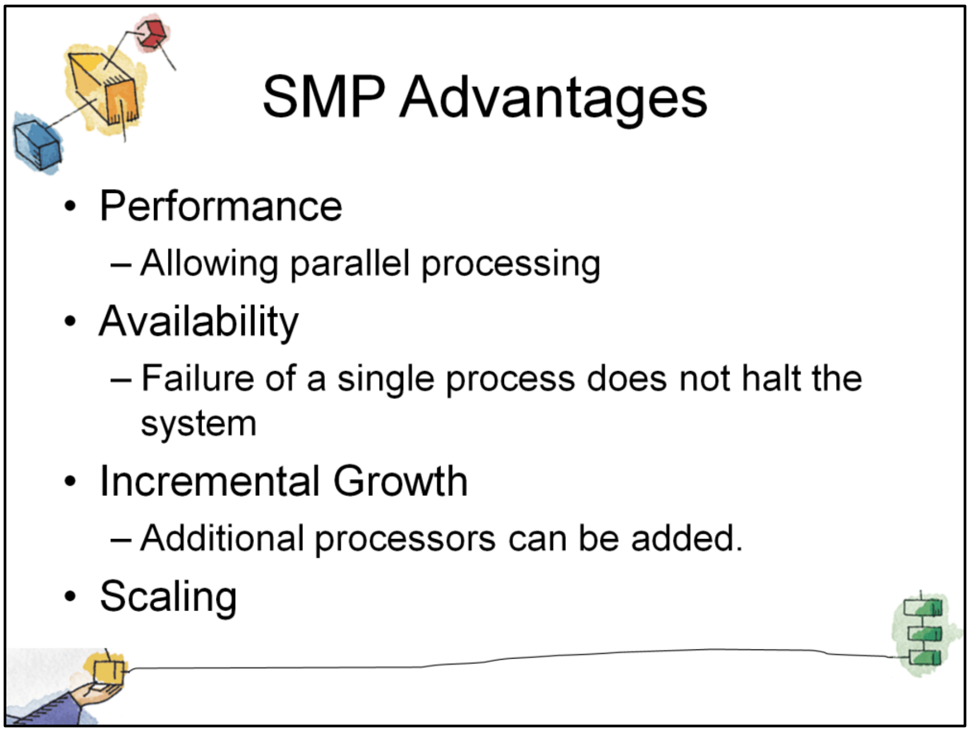
To achieve greater efficiency and reliability, one technique is to employ symmetric multiprocessing (SMP),

- SMP refers to a computer hardware architecture and also to the OS behavior that exploits that architecture.

A symmetric multiprocessor can be defined as a standalone computer system with the following characteristics:

1. There are multiple processors.
2. These processors share the same main memory and I/O facilities, interconnected by a communications bus or other internal connection scheme.
3. All processors can perform the same functions (hence the term symmetric).

The OS of an SMP schedules processes or threads across all of the processors.



SMP Advantages

- Performance
  - Allowing parallel processing
- Availability
  - Failure of a single process does not halt the system
- Incremental Growth
  - Additional processors can be added.
- Scaling

SMP has a number of potential advantages over uniprocessor architecture, including the following:

Performance:

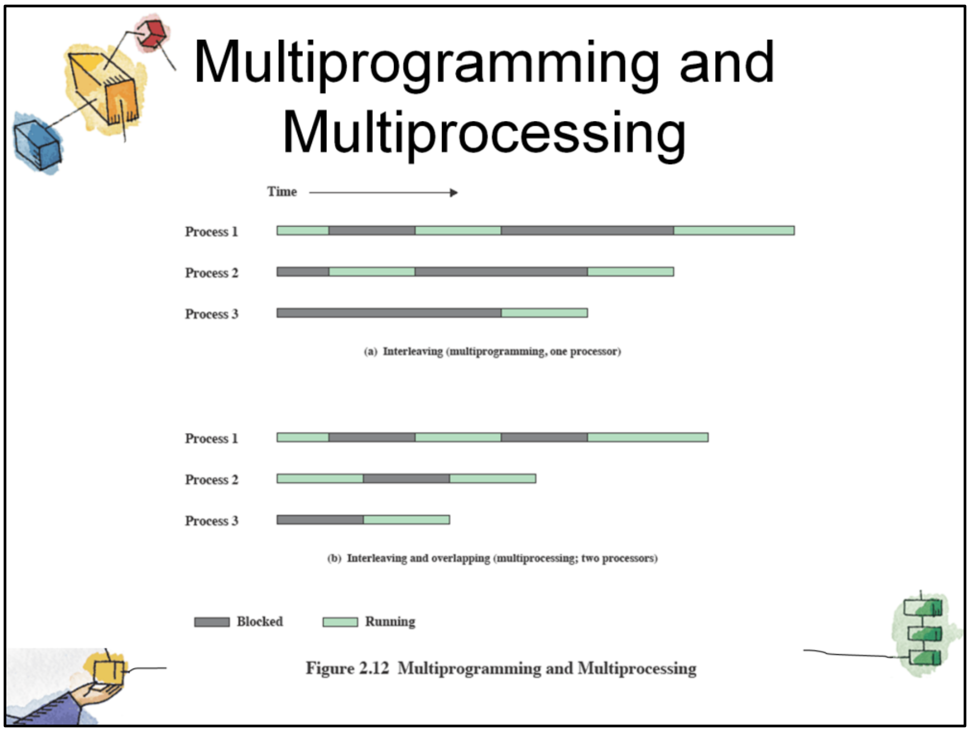
- If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.

Availability: In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the system.

- Instead, the system can continue to function at reduced performance.

Incremental growth: A user can enhance the performance of a system by adding an additional processor.

Scaling: Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.



Multiprogramming and Multiprocessing

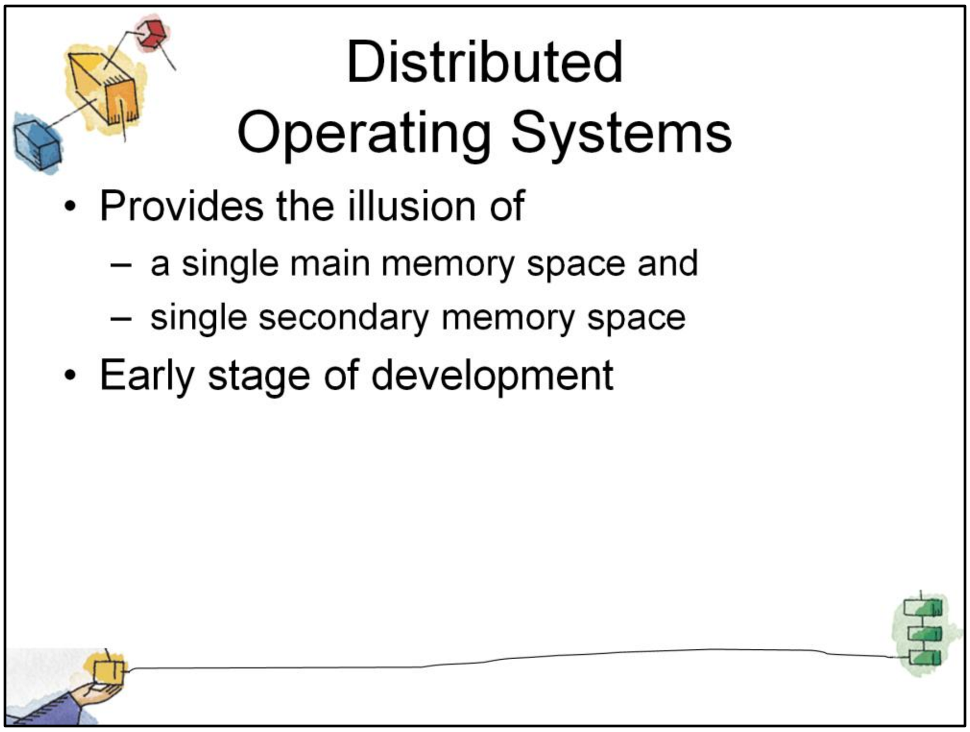
Time
Process 1
Process 2
Process 3
(a) Interleaving (multiprogramming, one processor)
Process 1
Process 2
Process 3
(b) Interleaving and overlapping (multiprocessing; two processors)
Blocked
Running

Figure 2.12 Multiprogramming and Multiprocessing

With multiprogramming, only one process can execute at a time;

- meanwhile all other processes are waiting for the processor.

With multiprocessing, more than one process can be running simultaneously, each on a different processor.



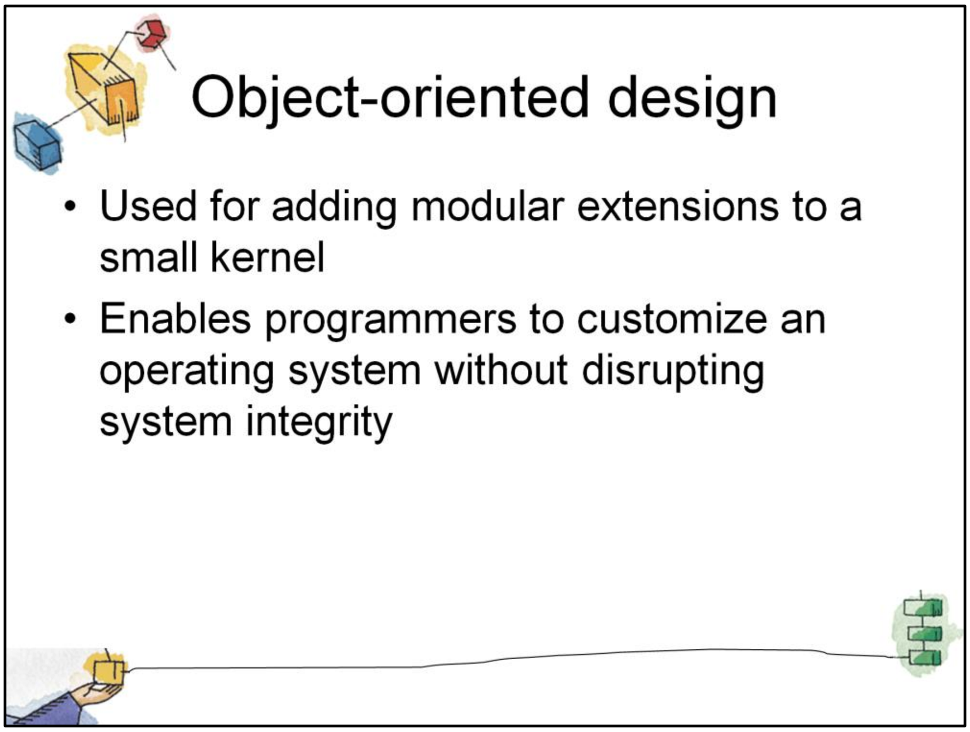
Distributed Operating Systems

- Provides the illusion of
  - a single main memory space and
  - single secondary memory space
- Early stage of development

A distributed operating system provides the illusion of a single main memory space and a single secondary memory space, plus other unified access facilities, such as a distributed file system.

Although clusters are becoming increasingly popular, the state of the art for distributed operating systems lags that of uniprocessor

and SMP operating systems.



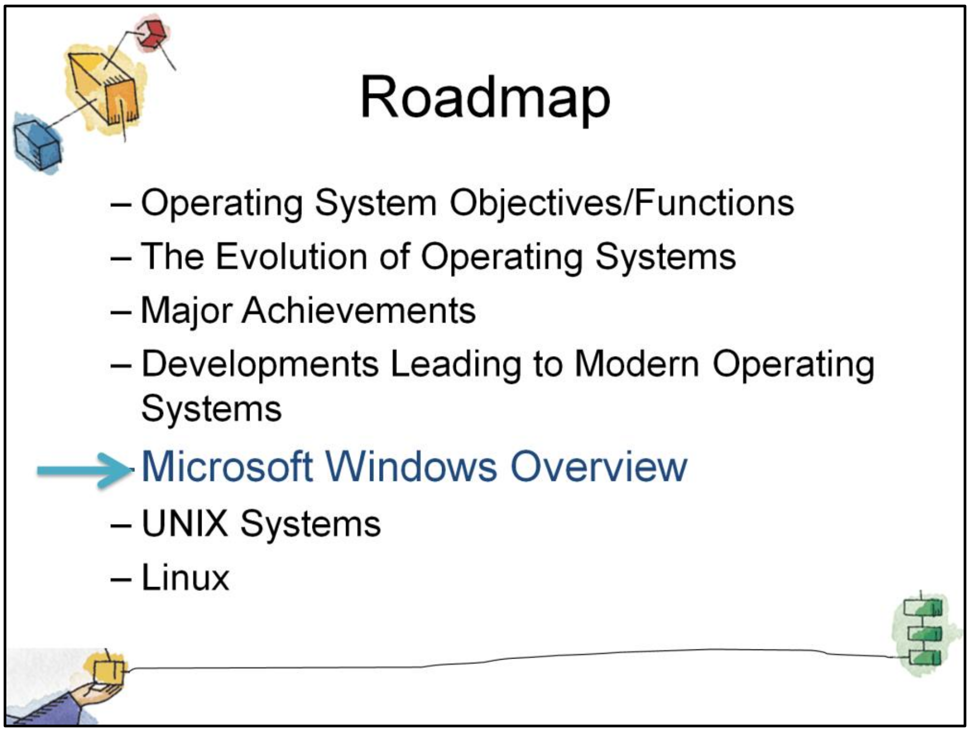
Object-oriented design

- Used for adding modular extensions to a small kernel
- Enables programmers to customize an operating system without disrupting system integrity

Object-oriented design lends discipline to the process of adding modular extensions to a small kernel.

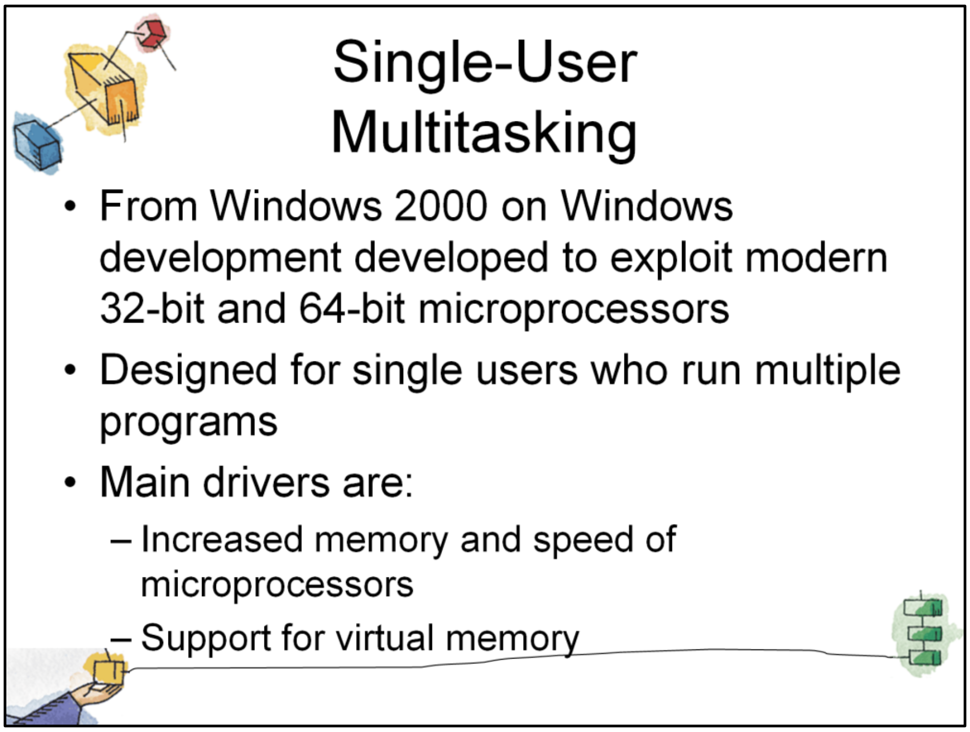
At the OS level, an object-based structure enables programmers to customize an OS without disrupting system integrity.

Object orientation also eases the development of distributed tools and full-blown distributed operating systems.



Roadmap

– Operating System Objectives/Functions
– The Evolution of Operating Systems
– Major Achievements
– Developments Leading to Modern Operating Systems
→ Microsoft Windows Overview
– UNIX Systems
– Linux



Single-User Multitasking

- From Windows 2000 on Windows development developed to exploit modern 32-bit and 64-bit microprocessors
- Designed for single users who run multiple programs
- Main drivers are:
  - Increased memory and speed of microprocessors
  - Support for virtual memory

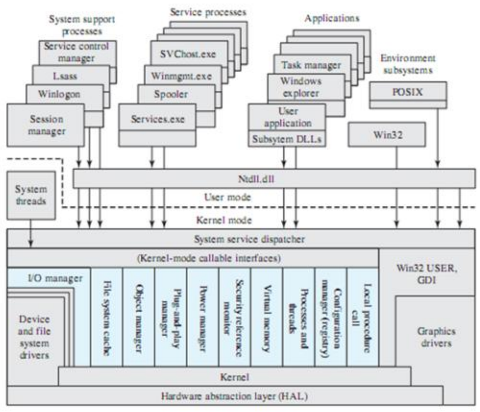
Windows was driven by a need to exploit the processing capabilities of today’s 32-bit and 64-bit microprocessors, which rival mainframes of just a few years ago in speed, hardware sophistication, and memory capacity.

One of the most significant features of these new operating systems is that, although they are still intended for support of a single interactive user, they are multitasking operating systems.

Two main developments have triggered the need for multitasking on personal computers, workstations, and servers.

- The increased speed and memory capacity of microprocessors,
- The support for virtual memory.

Windows Architecture



Lsass = local security authentication server
POSIX = portable operating system interface
GDI = graphics device interface
DLL = dynamic link libraries

Colored area indicates Executive

Figure 2.13 **Windows and Windows Vista Architecture [RUSS05]**

Figure 2.13 illustrates the overall structure of Windows 2000; later releases of Windows, including Vista, have essentially the same structure at this level of detail.

- Its modular structure gives Windows considerable flexibility.

Windows separates application-oriented software from the core OS software.

- The latter, which includes the Executive, the Kernel, device drivers, and the hardware abstraction layer, runs in kernel mode.

Kernel mode software has access to system data and to the hardware.

The remaining software, running in user mode, has limited access to system data.

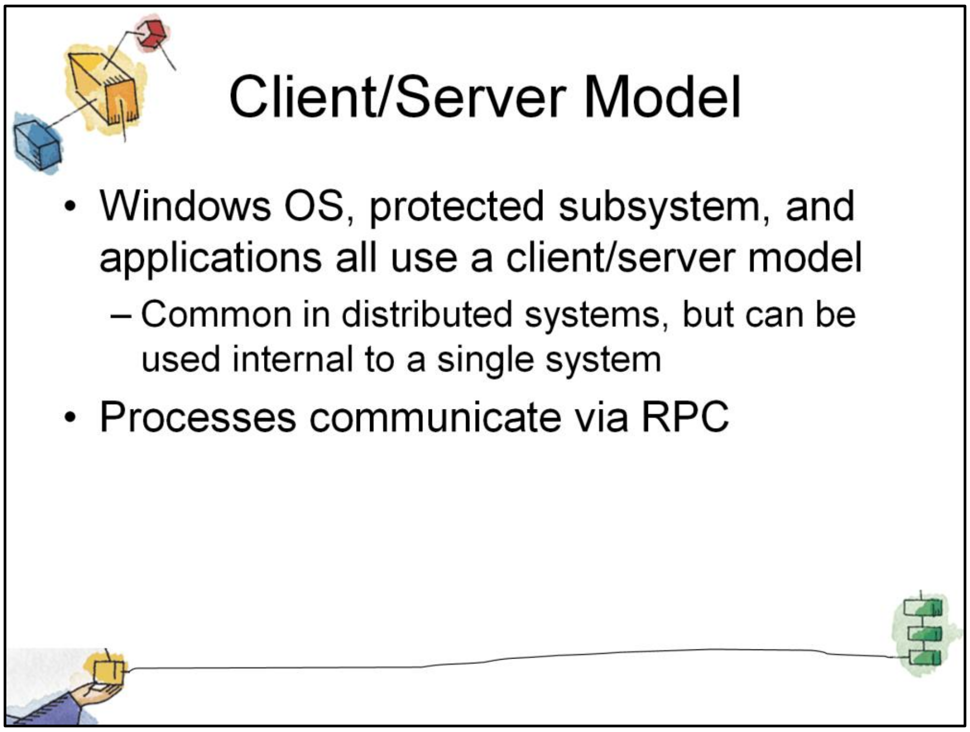
Windows has a highly modular architecture.

Each system function is managed by just one component of the OS.

The rest of the OS and all applications access that function through the responsible component using standard interfaces.

Key system data can only be accessed through the appropriate function.

In principle, any module can be removed, upgraded, or replaced without rewriting the entire system or its standard application program interface (APIs).



Client/Server Model

- Windows OS, protected subsystem, and applications all use a client/server model
  - Common in distributed systems, but can be used internal to a single system
- Processes communicate via RPC

The Windows operating system services, the protected subsystems, and the applications are structured using the client/server computing model, which is a common model for distributed computing

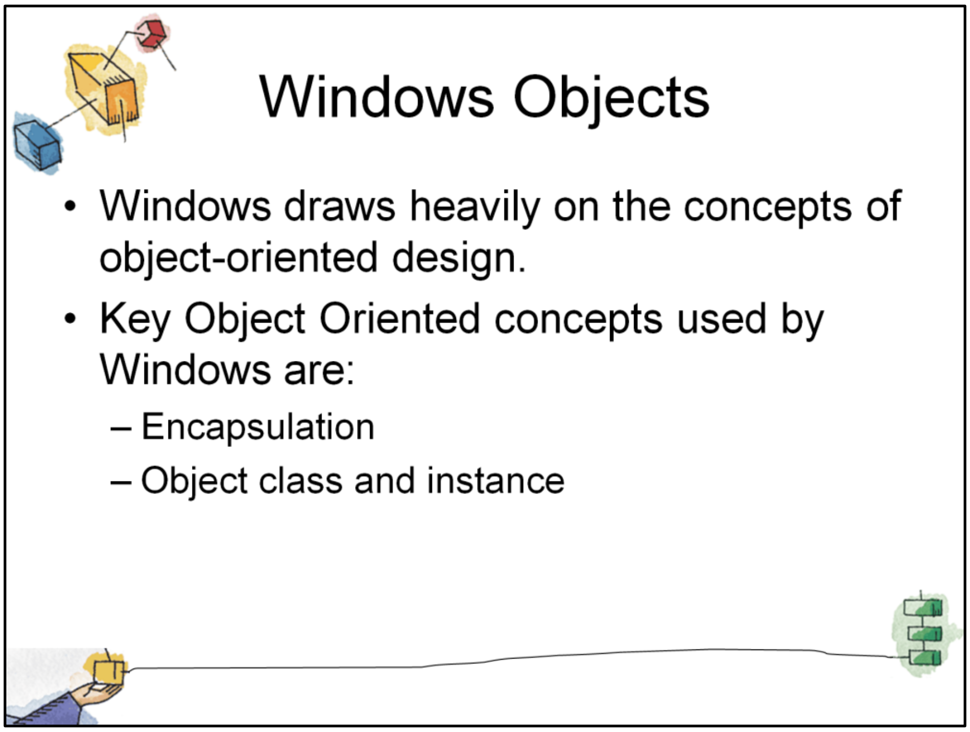
The native NT API is a set of kernel-based services which provide the core abstractions used by the system, such as processes, threads, virtual memory, I/O, and communication.

Windows provides a far richer set of services by using the client/server model to implement functionality in user-mode processes.

- Both the environment subsystems and the Windows user-mode services are implemented as processes that communicate with clients via RPC.
- Each server process waits for a request from a client for one of its services (for example, memory services, process creation services, or networking services).

A client, which can be an application program or another server program, requests a service by sending a message.

- The message is routed through the Executive to the appropriate server.
- The server performs the requested operation and returns the results or status information by means of another message, which is routed through the Executive back to the client.



Windows Objects

- Windows draws heavily on the concepts of object-oriented design.
- Key Object Oriented concepts used by Windows are:
  - Encapsulation
  - Object class and instance

Windows draws heavily on the concepts of object-oriented design.

This approach facilitates the sharing of resources and data among processes and the protection of resources from unauthorized access.

Among the key object-oriented concepts used by Windows are the following:

Encapsulation: An object consists of one or more items of data, called attributes, and one or more procedures that may be performed on those data, called services.

- The only way to access the data in an object is by invoking one of the object's services.
- Thus, the data in the object can easily be protected from unauthorized use and from incorrect use (e.g., trying to execute a nonexecutable piece of data).

Object class and instance: An object class is a template that lists the attributes and services of an object and defines certain object characteristics.

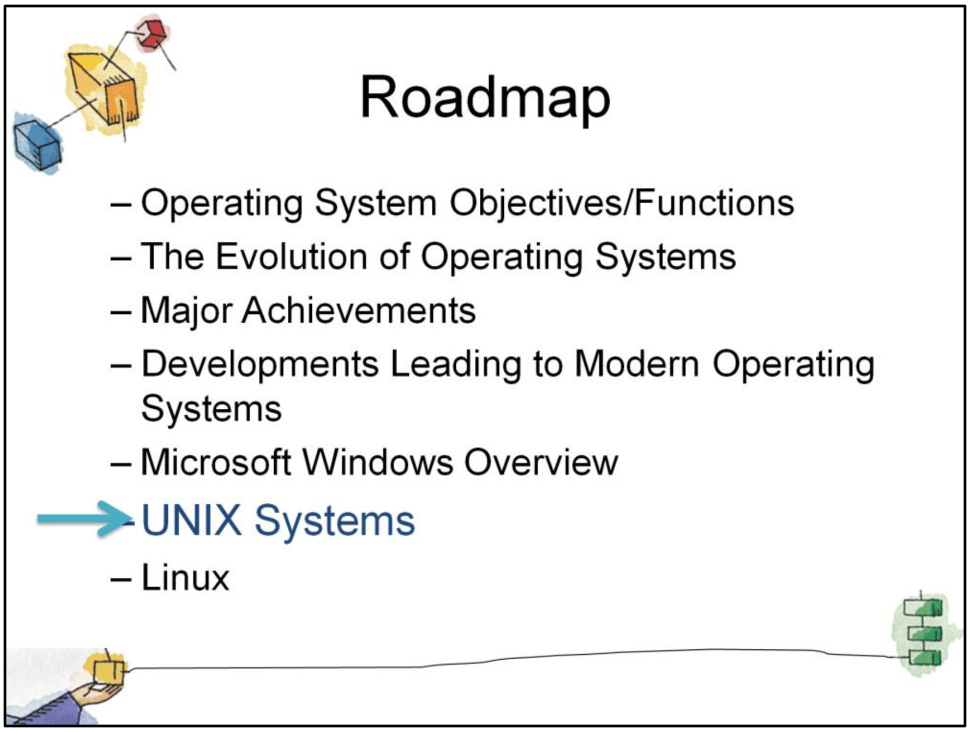
- The OS can create specific instances of an object class as needed.
- e.g., there is a single process object class and one process object for every currently active process.
- This approach simplifies object creation and management.

Inheritance: Although the implementation is hand coded, the Executive uses inheritance to extend object classes by adding new features.

- Every Executive class is based on a base class which specifies virtual methods that support creating, naming, securing, and deleting objects.
- Dispatcher objects are Executive objects that inherit the properties of an event object, so they can use common synchronization methods.
- Other specific object types, such as the device class, allow classes for specific devices to inherit from the base class, and add additional data and methods.

Polymorphism: Internally, Windows uses a common set of API functions to manipulate objects of any type; this is a feature of polymorphism.

- However, Windows is not completely polymorphic because there are many APIs that are specific to specific object types.



Roadmap

- Operating System Objectives/Functions
- The Evolution of Operating Systems
- Major Achievements
- Developments Leading to Modern Operating Systems
- Microsoft Windows Overview
- UNIX Systems
- Linux

Description of UNIX

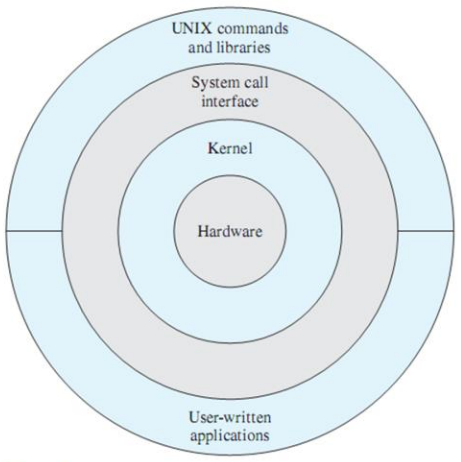


Figure 2.14 **General UNIX Architecture**

This figure provides a general description of the classic UNIX architecture.

The underlying hardware is surrounded by the OS software.

The OS is often called the system kernel, or simply the kernel, to emphasize its isolation from the user and applications.

It is the UNIX kernel that we will be concerned with in our use of UNIX as an example in this book.

UNIX also comes equipped with a number of user services and interfaces that are considered part of the system.

- These can be grouped into the shell, other interface software, and the components of the C compiler (compiler, assembler, loader).
- The layer outside of this consists of user applications and the user interface to the C compiler.

Traditional UNIX Kernel

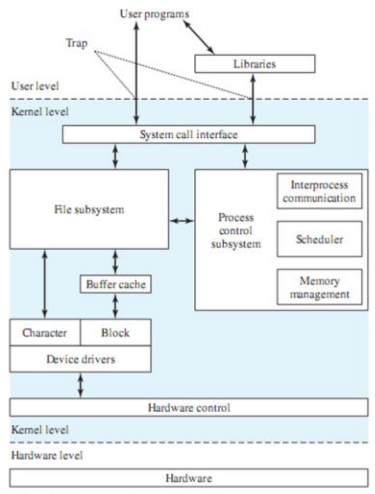


Figure 2.15 Traditional UNIX Kernel

User programs can invoke OS services either directly or through library programs.

The system call interface is the boundary with the user and allows higher-level software to gain access to specific kernel functions.

At the other end, the OS contains primitive routines that interact directly with the hardware.

Between these two interfaces, the system is divided into two main parts, one concerned with process control and the other concerned with file management and I/O.

- The process control subsystem is responsible for memory management, the scheduling and dispatching of processes, and the synchronization and interprocess communication of processes.
- The file system exchanges data between memory and external devices either as a stream of characters or in blocks. To achieve this, a variety of device drivers are used.
    - For block-oriented transfers, a disk cache approach is used: a system buffer in main memory is interposed between the user address space and the external device.

System V Release 4 (SVR4)

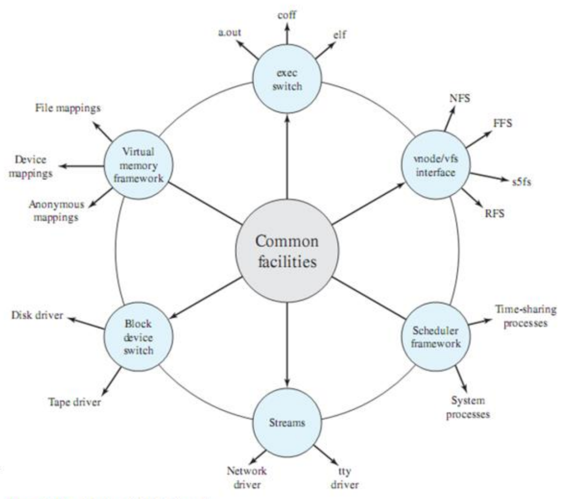
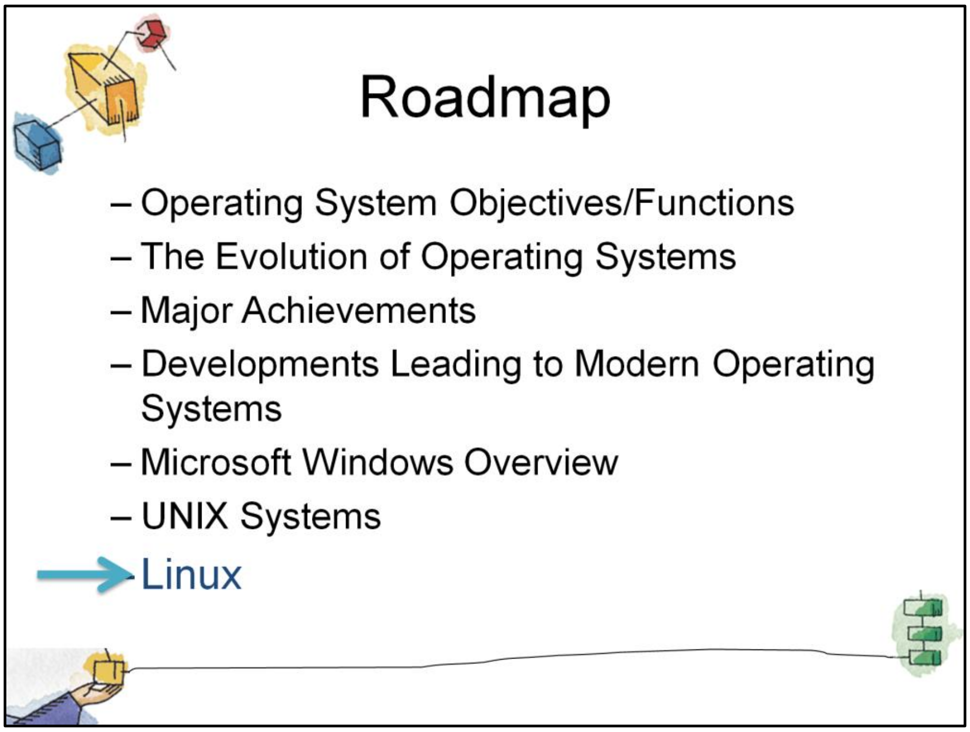


Figure 2.16 **Modern UNIX Kernel**

Almost a complete rewrite of the System V kernel and produced a clean, if complex, implementation.

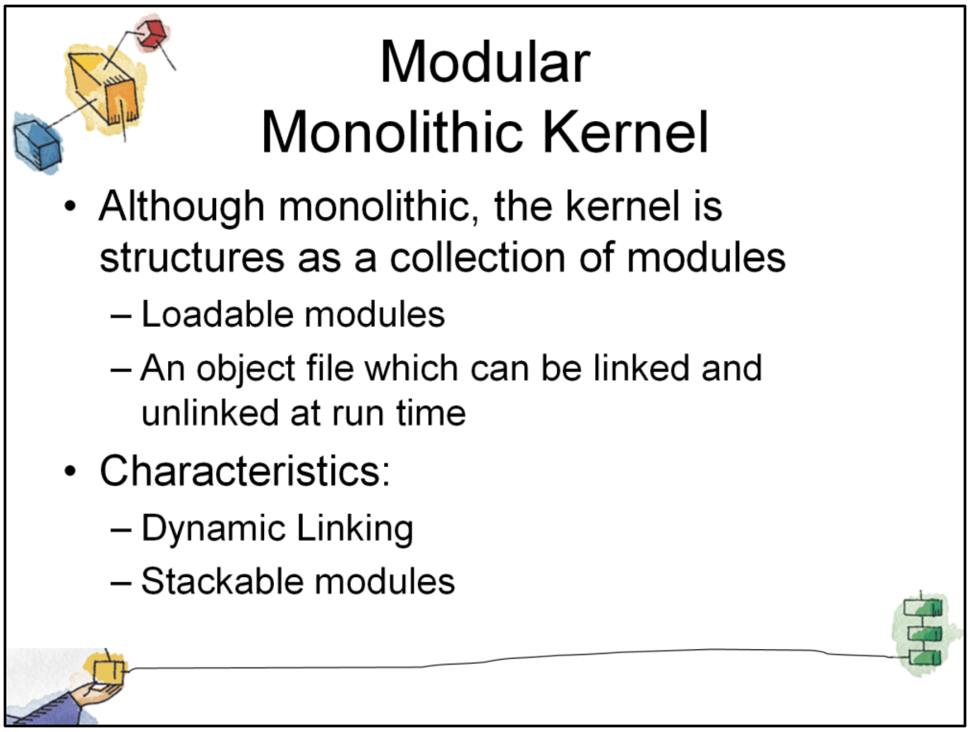
New features in the release include real-time processing support, process scheduling classes, dynamically allocated data structures, virtual memory management, virtual file system, and a preemptive kernel.

It incorporates most of the important features ever developed on any UNIX system and does so in an integrated, commercially viable fashion.



Roadmap

– Operating System Objectives/Functions
– The Evolution of Operating Systems
– Major Achievements
– Developments Leading to Modern Operating Systems
– Microsoft Windows Overview
– UNIX Systems
→ Linux



Modular Monolithic Kernel

- Although monolithic, the kernel is structures as a collection of modules
  - Loadable modules
  - An object file which can be linked and unlinked at run time
- Characteristics:
  - Dynamic Linking
  - Stackable modules

Although Linux does not use a microkernel approach, it achieves many of the potential advantages of this approach by means of its particular modular architecture.

Linux is structured as a collection of modules, a number of which can be automatically loaded and unloaded on demand.

- These relatively independent blocks are referred to as loadable modules.
- In essence, a module is an object file whose code can be linked to and unlinked from the kernel at runtime.

The Linux loadable modules have two important characteristics:

Dynamic linking: A kernel module can be loaded and linked into the kernel while the kernel is already in memory and executing.

- A module can also be unlinked and removed from memory at any time.
- Saves kernel memory.

Stackable modules: The modules are arranged in a hierarchy.

- Individual modules serve as libraries when they are referenced by client modules higher up in the hierarchy, and as clients when they reference modules further down.

Linux Kernel Modules

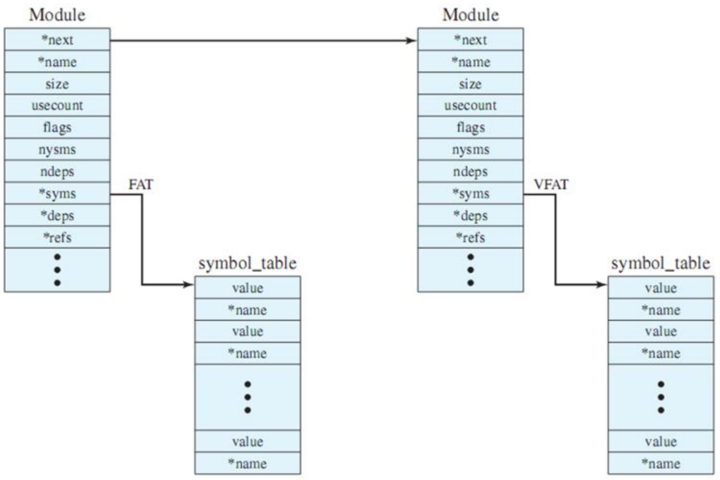


Figure 2.17 **Example List of Linux Kernel Modules**

Figure 2.17 is an example that illustrates the structures used by Linux to manage modules.
It shows that the VFAT module was loaded after the FAT module and that the VFAT module is dependent on the FAT module.

The figure shows the list of kernel modules after only two modules have been loaded:
- FAT and VFAT.

Each module is defined by two tables,
- The module table and
- the symbol table.

The module table includes the following elements:
- *next: Pointer to the following module. All modules are organized into a linked list.
    - The list begins with a pseudomodule (not shown in Figure 2.17).
- *name: Pointer to module name.
- size:Module size in memory pages.
- usecount:Module usage counter.
    - The counter is incremented when an operation involving the module’s functions is started and decremented when the operation terminates.
- flags:Module flags.
- nsyms: Number of exported symbols.
- ndeps: Number of referenced modules
- *syms: Pointer to this module’s symbol table.
- *deps: Pointer to list of modules the are referenced by this module.
- *refs: Pointer to list of modules that use this module.

The symbol table defines those symbols controlled by this module that are used elsewhere.

Linux Kernel Components

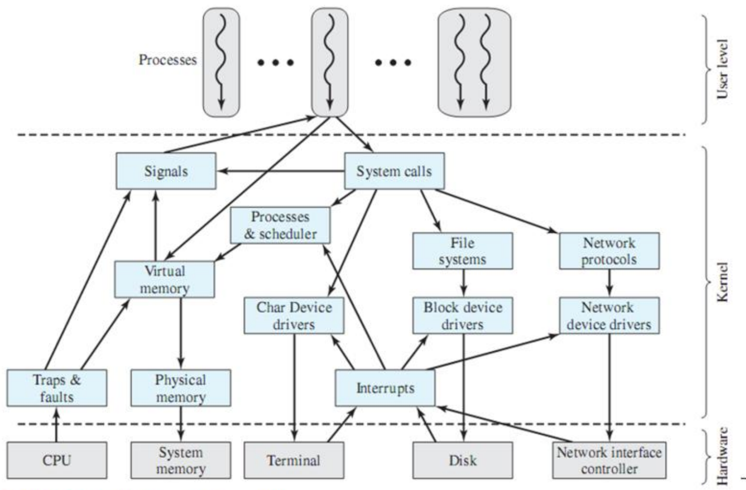


Figure 2.18 Linux Kernel Components

This figure shows the main components of the Linux kernel as implemented on an IA-64 architecture (e.g., Intel Itanium).

It shows several processes running on top of the kernel.

Each box indicates a separate process, while each squiggly line with an arrowhead represents a thread of execution.

The kernel itself consists of an interacting collection of components, with arrows indicating the main interactions.

- The underlying hardware is also depicted as a set of components with arrows indicating which kernel components use or control which hardware components.

All of the kernel components, of course, execute on the processor but, for simplicity, these relationships are not shown.