Operating Systems

Introduction

- Knowledge of methods of prevention and recovery from a system deadlock.

Unit	Details		Lectures
I	**Operating System Overview:** Objectives and Functions, Evolution, Achievements, Modern Operating Systems, Fault tolerance, OS design considerations for multiprocessor and multicore, overview of different operating systems **Processes:** Process Description and Control.		**12**
II	**Threads, Concurrency:** Mutual Exclusion and Synchronization.		**12**
III	**Concurrency:** Deadlock and Starvation, **Memory:** Memory Management, Virtual Memory.		**12**
IV	**Scheduling:** Uniprocessor Scheduling, Multiprocessor and Real-Time Scheduling		**12**
V	**IO and File Management:** I/O Management and Disk Scheduling, File Management, **Operating System Security.**		**12**

Books and References:

Sr. No.	Title	Author/s	Publisher	Edition	Year
1.	Operating Systems – Internals and Design Principles	Willaim Stallings	Pearson	9th	2009
2.	Operating System Concepts	Abraham Silberschatz,	Wiley	8th	

Operating systems are those programs that interface the machine with the applications programs. The main function of these systems is to dynamically allocate the shared system resources to the executing programs. As such, research in this area is clearly concerned with the management and scheduling of memory, processes, and other devices. But the interface with adjacent levels continues to shift with time. Functions that were originally part of the operating system have migrated to the hardware. On the other side, programmed functions extraneous to the problems being solved by the application programs are included in the operating system.

—*WHAT CAN BE AUTOMATED?: THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY*, MIT PRESS, 1980

2.1 OPERATING SYSTEM OBJECTIVES AND FUNCTIONS

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.

Let us examine these three aspects of an OS in turn.

The Operating System as a User/Computer Interface

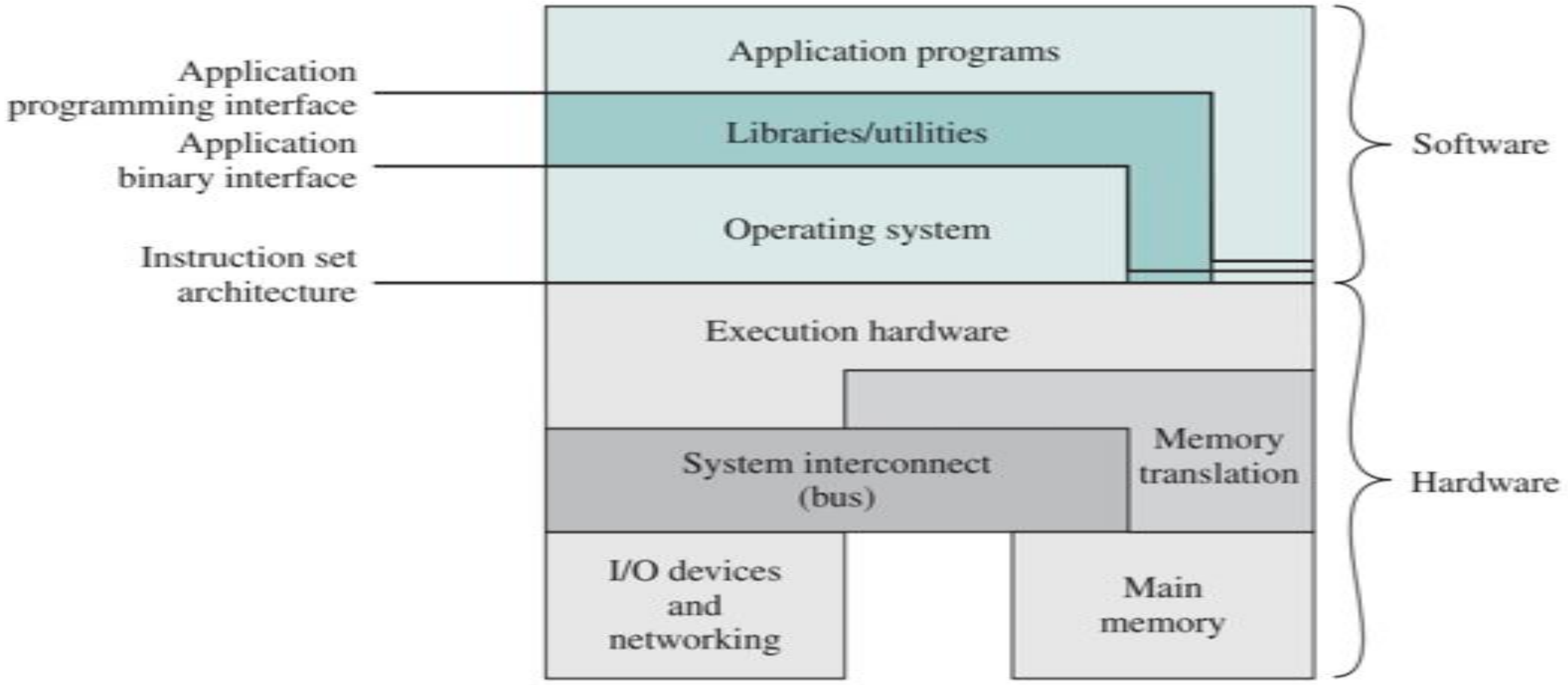

Application programs
Application programming interface
Libraries/utilities
Application binary interface
Operating system
Instruction set architecture
Software
Execution hardware
Memory translation
System interconnect (bus)
Hardware
I/O devices and networking
Main memory

Figure 2.1 **Computer Hardware and Software Structure**

The Operating System as Resource Manager

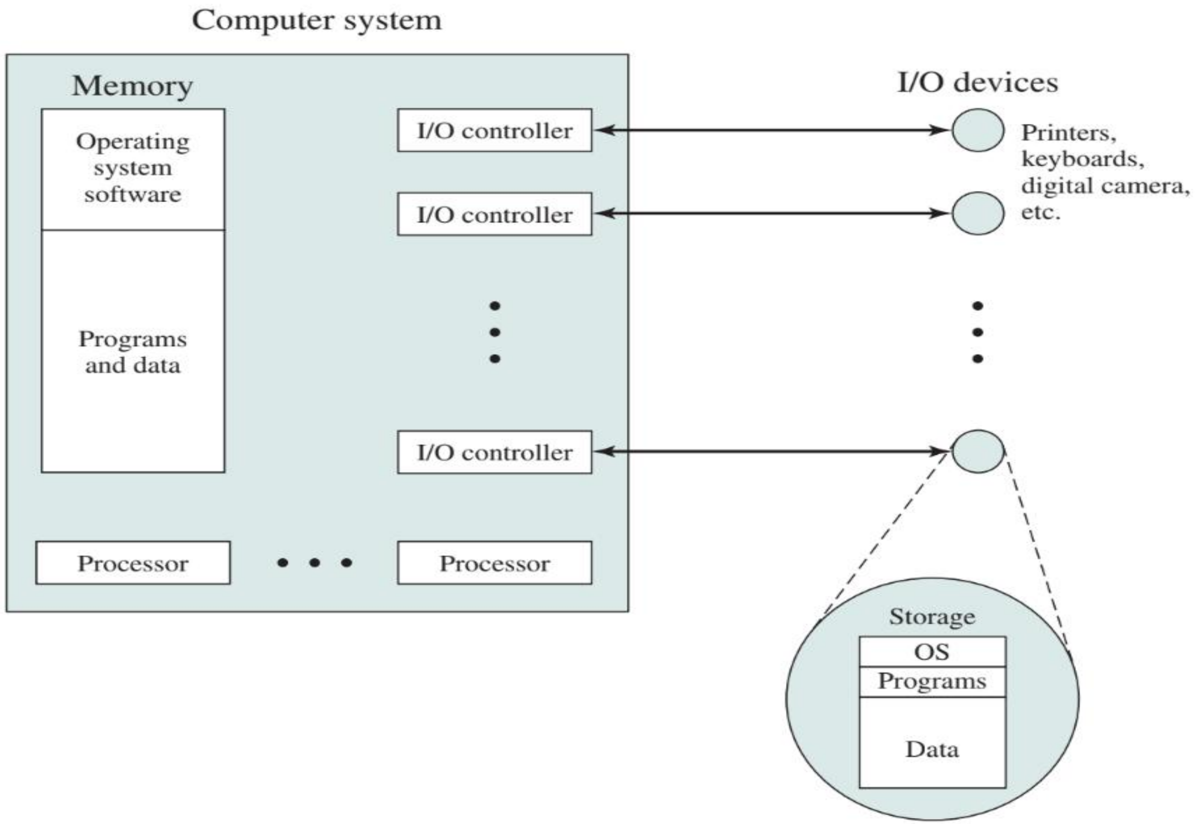


Figure 2.2 **The Operating System as Resource Manager**

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

The OS functions in the same way as ordinary computer software; that is, 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.

This includes the **kernel** , or **nucleus** , which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The memory management hardware in the processor and the OS jointly control the allocation of main memory. 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.

MAJOR ACHIEVEMENTS

Operating systems are among the most complex pieces of software ever developed. This reflects the challenge of trying to meet the difficult and in some cases competing objectives of convenience, efficiency, and ability to evolve. [DENN80a] proposes that there have been four major theoretical advances in the development of operating systems:

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

Each advance is characterized by principles, or abstractions, developed to meet difficult practical problems. Taken together, these **five areas span many of the key design and implementation issues of modern operating systems**. The brief review of these five areas in this section serves as an overview of much of the rest of the text.

The Process

Central to the design of operating systems is the concept of process. **A process is a program in execution.** Three major lines of computer system development created problems in timing and synchronization that contributed to the development of the concept of the process: **multiprogramming batch operation, time sharing, and real-time transaction systems.**

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.

Causes of errors in OS design:

Improper synchronization:

Failed mutual exclusion:

Non determinate program operation:

Deadlocks: It is possible for two or more programs to be hung up waiting for each other.

Memory Management

Process isolation:

Automatic allocation and management:

Support of modular programming:

Protection and access control:

Long-term storage:

Typically, operating systems meet these 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.

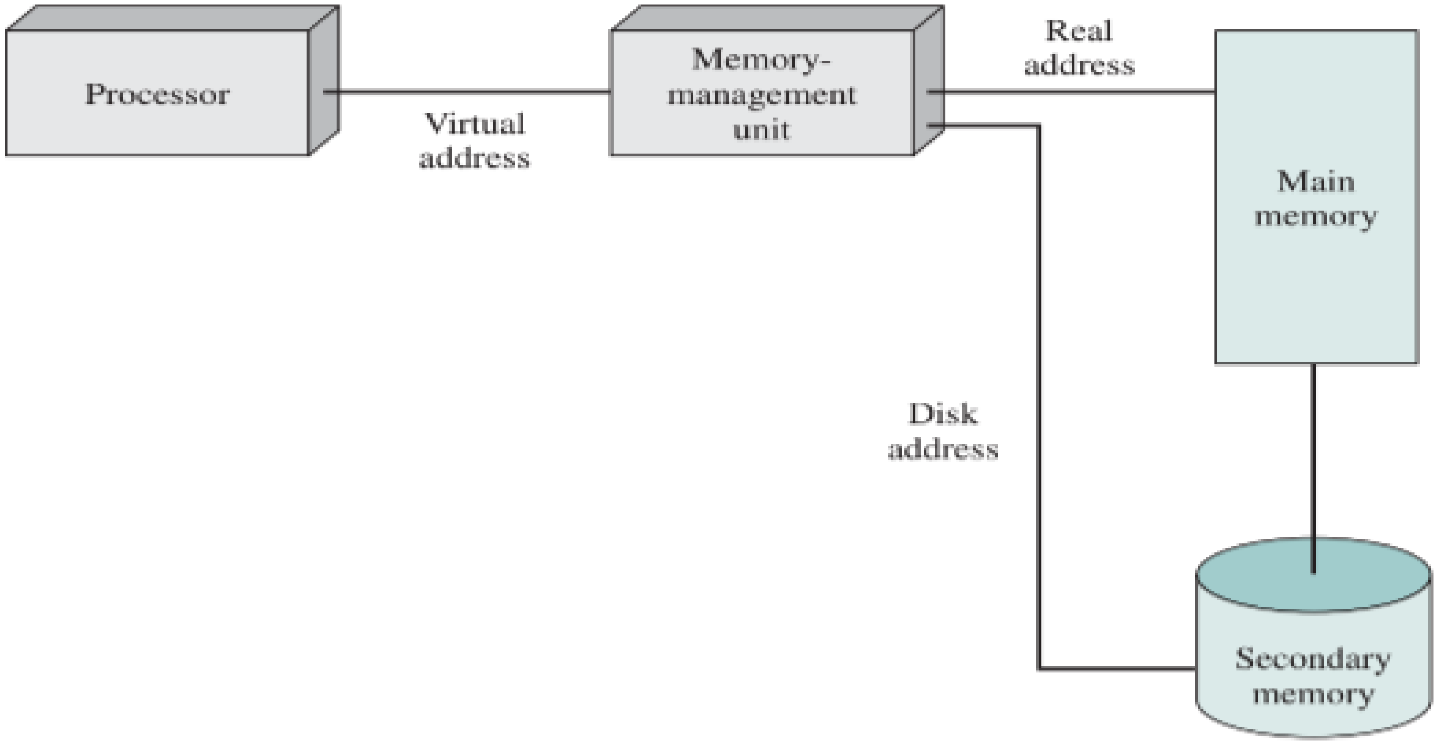


Figure 2.10 **Virtual Memory Addressing**

Information Protection and Security

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

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.

Scheduling and Resource Management

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. For example, 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.

Scheduling and Resource Management

Efficiency: The OS should attempt to maximize throughput, minimize response time, and, in the case of time sharing, accommodate as many users as possible. These criteria conflict; finding the right balance for a particular situation is an ongoing problem for OS research.

THE EVOLUTION OF OPERATING SYSTEMS

Serial Processing

The mode of operation could be termed serial processing , reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed to attempt to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and I/O driver routines that were available as common software for all users.

THE EVOLUTION OF OPERATING SYSTEMS

Serial Processing

These early systems presented two main problems:

Scheduling:

Setup time:

Simple Batch Systems

To improve utilization, the concept of a batch OS was developed. It appears that the first batch OS (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701 [WEIZ81].

THE EVOLUTION OF OPERATING SYSTEMS

Simple Batch Systems

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.

Simple Batch Systems

The monitor controls the sequence of events. For this to be so, much of the monitor must always be in main memory and available for execution. 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.

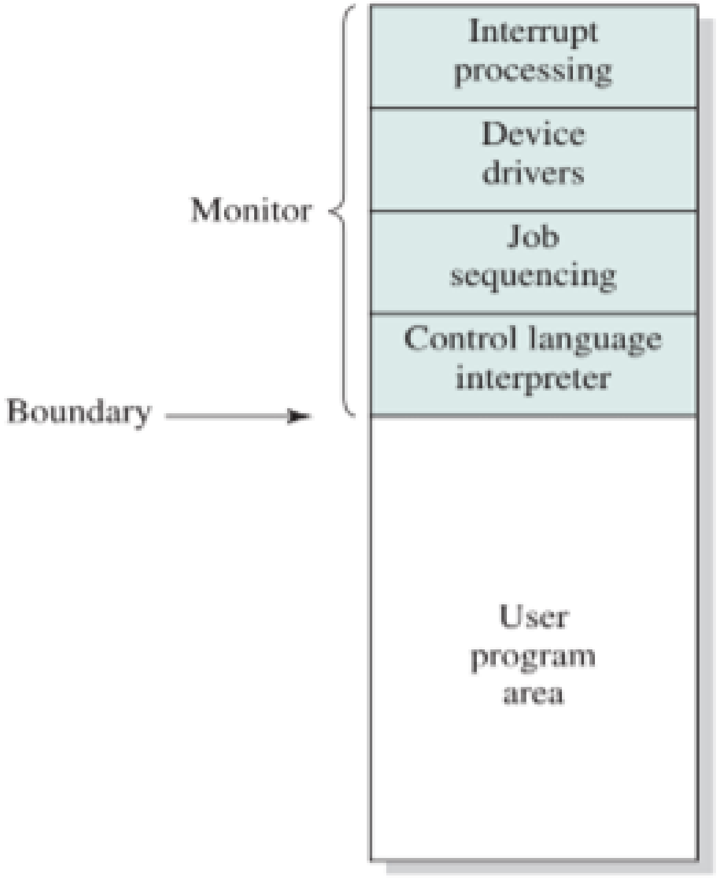


Figure 2.3 Memory Layout for a Resident Monitor

Multiprogrammed Batch Systems

Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle. The problem is that I/O devices are slow compared to the processor.

Memory can be expanded to hold three, four, or more
programs and switch among all of them. The approach is known as **multiprogramming , or multitasking** . It is the central theme of modern operating systems.

Multiprogramming operating systems are fairly sophisticated compared to single-program, or uniprogramming , systems. To have several jobs ready to run, they must be kept in main memory, requiring some form of memory management .

In addition, if several jobs are ready to run, the processor must decide which one to run, this decision requires an **algorithm for scheduling.**

Time-Sharing Systems

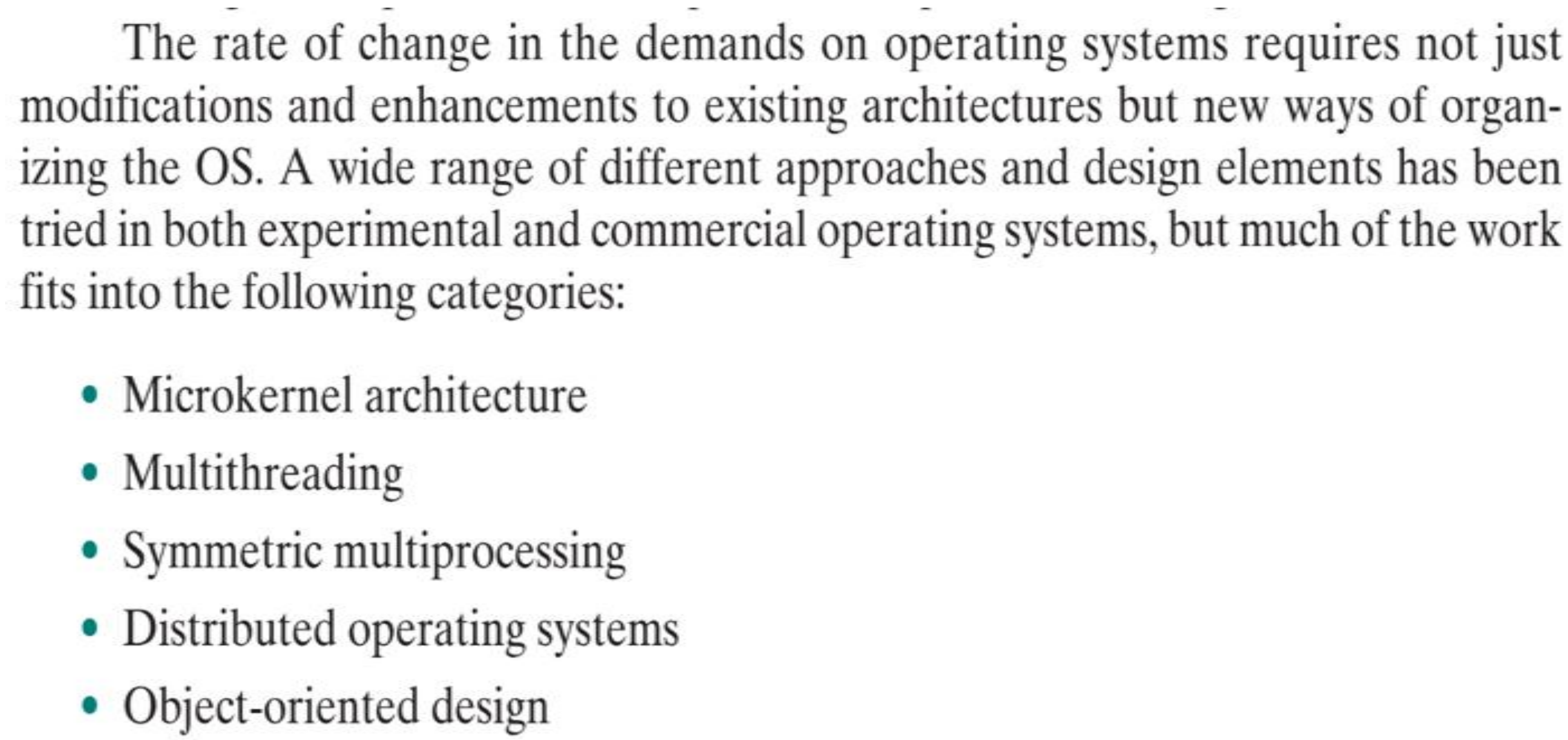
With the use of multiprogramming, batch processing can be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. This technique is referred to as time sharing , because processor time is shared among multiple users.

Time-Sharing Systems

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.

DEVELOPMENTS LEADING TO MODERN OPERATING SYSTEMS



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

Monolithic kernel

It is thought of as OS functionality is provided in the 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.

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

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 interruptable 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.

With multiple threads running within the same process, switching back and forth among threads involves less processor overhead than a major process switch between different processes. Threads are also useful for structuring processes that are part of the OS kernel.

A **distributed system** in its most simplest definition is a group of computers working together as to appear as a single computer to the end-user.
A collection of entities (computers), each with its own main memory, secondary memory, and other I/O modules. A distributed operating system provides the illusion of a single main memory space and a single secondary memory space.

Another innovation in OS design is the use of object-oriented technologies. **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.

DESIGN CONSIDERATIONS FOR MULTIPROCESSOR AND MULTICORE

Symmetric Multiprocessor OS Considerations

In an SMP system, the kernel can execute on any processor, and typically each processor does self-scheduling from the pool of available processes or threads.

The kernel can be constructed as multiple processes or multiple threads, allowing portions of the kernel to execute in parallel.

The SMP approach **complicates** the OS. The OS designer must deal with the **complexity due to sharing resources (like data structures) and coordinating actions** (like accessing devices) from multiple parts of the OS executing at the same time. **Techniques must be employed to resolve and synchronize claims to resources.**

An SMP operating system manages processor and other computer resources so that the **user may view the system in the same fashion as a multiprogramming uniprocessor system.**

A multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors. The key design issues include the following:

Simultaneous concurrent processes or threads: With multiple processors executing the same or different parts of the kernel, kernel tables and management structures must be managed properly to avoid data corruption or invalid operations.

Scheduling: Any processor may perform scheduling, which complicates the task of enforcing a scheduling policy and assuring that corruption of the scheduler data structures is avoided.

Synchronization: With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization. Synchronization is a facility that enforces mutual exclusion and event ordering. A common synchronization mechanism used in multiprocessor operating systems is **locks**.

Memory management: Memory management on a multiprocessor must deal with all of the issues found on uniprocessor computers and the OS needs to exploit the available hardware parallelism to achieve the best performance. The **paging mechanisms** on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement.

Reliability and fault tolerance: The OS should provide graceful degradation in the face of processor failure. The scheduler and other portions of the OS must recognize the loss of a processor and restructure management tables accordingly.

Fault tolerance: Fault tolerance is a process that enables an operating system to respond to a failure in hardware or software. This fault-tolerance definition refers to the **system's ability to continue operating despite failures or malfunctions.**

That is the process of working of a system in a proper way in spite of the occurrence of the failures in the system.

Multicore OS Considerations

The considerations for multicore systems **include all the design issues discussed so far** in this section for SMP systems. But additional concerns arise. **The issue is one of the scale of the potential parallelism.** Current multicore vendors offer systems with up to **eight cores** on a single chip. With each succeeding processor technology generation, the number of cores and the amount of shared and dedicated cache memory increases, so that we are now entering the era of “many-core” systems.

The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently. A central concern is how to match the inherent parallelism of a many-core system with the performance requirements of applications.

In essence, then, since the advent of **multicore** technology, OS designers have been struggling with the problem of how best to extract parallelism from computing workloads. **A variety of approaches are being explored for next-generation operating systems.**
Two general strategies:

PARALLELISM WITHIN APPLICATIONS

VIRTUAL MACHINE APPROACH

In essence, then, since the advent of multicore technology, OS designers have been struggling with the problem of how best to extract parallelism from computing workloads. **A variety of approaches are being explored for next-generation operating systems.**
Two general strategies:

PARALLELISM WITHIN APPLICATIONS
Most applications can, in principle, be subdivided into multiple tasks that can execute in parallel, with these tasks then being implemented as multiple processes, perhaps each with multiple threads. The difficulty is that the developer must decide **how to split up the application work into independently executable tasks.** That is, the developer must decide what pieces can or should be executed asynchronously or in parallel. It is primarily the compiler and the programming language features that support the parallel programming design process. But, the OS can support this design process, at minimum, by efficiently allocating resources among parallel tasks as defined by the developer.

Perhaps the most effective initiative to support developers includes a **multicore support capability** known as **Grand Central Dispatch (GCD)**. GCD does not help the developer decide how to break up a task or application into separate concurrent parts. But once a developer has identified something that can be split off into a separate task, GCD makes it as **easy and noninvasive** as possible to actually do so.

In essence, GCD is a **thread pool mechanism**, in which the OS maps tasks onto threads representing an available degree of concurrency (plus threads for blocking on I/O). Windows also has a thread pool mechanism (since 2000), and thread pools have been heavily used in server applications for years.

GCD is hence not a major evolutionary step. **It is a new and valuable tool for exploiting the available parallelism of a multicore system.**

VIRTUAL MACHINE APPROACH

An alternative approach is to recognize that with the ever-increasing number of cores on a chip, the attempt to multiprogram individual cores to support multiple applications may be a misplaced use of resources. If instead, we allow one or more cores to be dedicated to a particular process and then leave the processor alone to devote its efforts to that process, we avoid much of the overhead of task switching and scheduling decisions.

The multicore OS could then act as a **hypervisor** that makes a high-level decision to allocate cores to applications but does little in the way of resource allocation beyond that.The programs themselves take on many of the duties of resource management. The OS assigns an application a processor and some memory, and the program itself, using metadata generated by the compiler, would best know how to use these resources.

VIRTUAL MACHINE APPROACH

An alternative approach is to recognize that with the ever-increasing number of cores on a chip, the attempt to multiprogram individual cores to support multiple applications may be a misplaced use of resources. If instead, we allow one or more cores to be dedicated to a particular process and then leave the processor alone to devote its efforts to that process, we avoid much of the overhead of task switching and scheduling decisions.

The multicore OS could then act as a **hypervisor** that makes a high-level decision to allocate cores to applications but does little in the way of resource allocation beyond that.The programs themselves take on many of the duties of resource management. The OS assigns an application a processor and some memory, and the program itself, using metadata generated by the compiler, would best know how to use these resources.

The Modern OS

Modern operating systems, such as today’s **Windows and UNIX (with all its flavors like Solaris, Linux, and MacOS X)**, must exploit the capabilities of all the billions of transistors on each silicon chip. They must work with **multiple 32-bit** and **64-bit CPUs**, with adjunct **GPUs, DSPs**, and fixed function units. They must provide **support for sophisticated input/output** (multiple touch-sensitive displays, cameras, microphones, biometric and other sensors) and **handle a variety of data challenges** (streaming media, photos, scientific number crunching, search queries)—all while giving a human being a **responsive, real-time experience with the computing system**.

The OS can must aggressively manage the system and coordinate between all applications, the competing computations that are taking place often simultaneously on the multiple CPUs, GPUs, and DSPs that may be present in a modern computing environment. Thus all modern operating systems have **multitasking capability.**

OVERVIIEW OF DIFFERENT OPERATING SYSTEMS

WINDOWS 7 THREAD AND SMP MANAGEMENT

History

The story of Windows begins with a very different OS, developed by Microsoft for the first IBM personal computer and referred to as MS-DOS.Microsoft's initial OS ran a single application at a time, using a command line interface to control the system. It took a long time for Microsoft to develop a true GUI interface for the PC; on their third try they succeeded.

Architecture

As with virtually all operating systems, Windows separates application-oriented software from the core OS software. 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. In principle, any module can be removed, upgraded, or replaced without rewriting the entire system or its standard application program interfaces (APIs).

WINDOWS 7 THREAD AND SMP MANAGEMENT

Two important characteristics of Windows are its **support for threads** and for **symmetric multiprocessing (SMP)**.

Though the core of Windows is written in C, the design principles followed draw 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 encapsulation, object class and instance, inheritance, polymorphism.

The core architecture of Windows has been very stable; however, at each release there are new features and improvements made even at the lower levels of the system. Many of the changes in Windows are not visible in the features themselves, but in the performance and stability of the system. These are due to changes in the engineering behind Windows.

Other improvements are due to new features, or improvements to existing features such as engineering improvements, performance improvements, reliability improvements, energy efficiency, security, thread improvements.

TRADITIONAL UNIX SYSTEMS

History

UNIX was initially developed at Bell Labs and became operational on a PDP-7 in 1970. The first notable milestone was porting the UNIX system from the PDP-7 to the PDP-11. This was the first hint that UNIX would be an OS for all computers. The next important milestone was the rewriting of UNIX in the programming language C. This was an unheard-of strategy at the time. The C implementation demonstrated the advantages of using a high-level language for most if not all of the system code. Today, virtually all UNIX implementations are written in C.

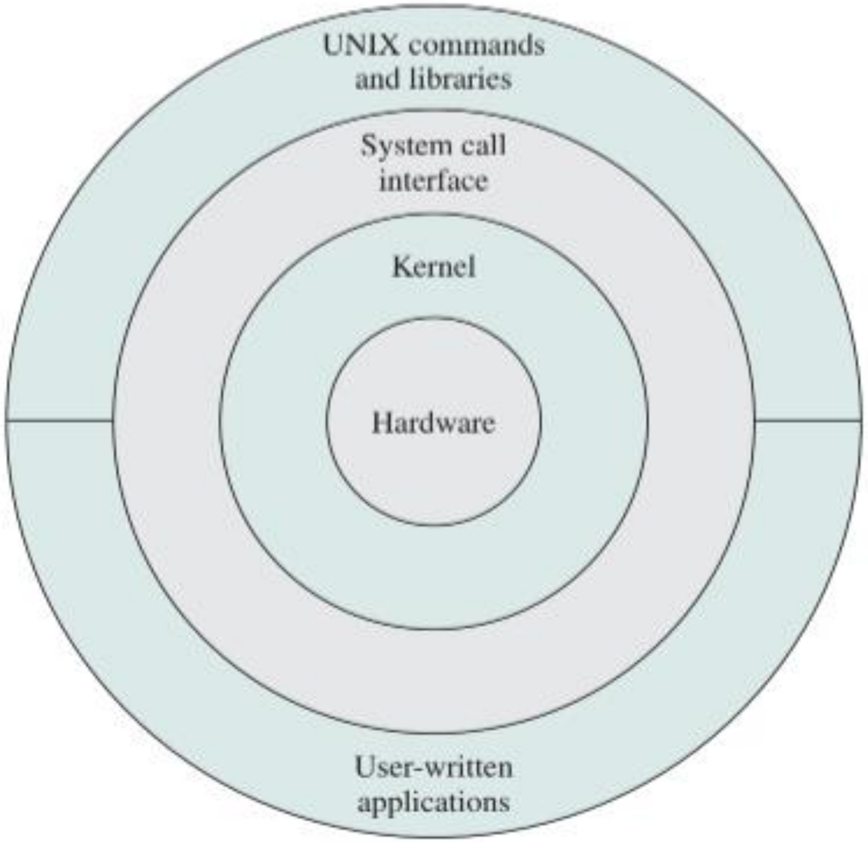


Figure 2.16 **General UNIX Architecture**

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.

MODERN UNIX SYSTEMS

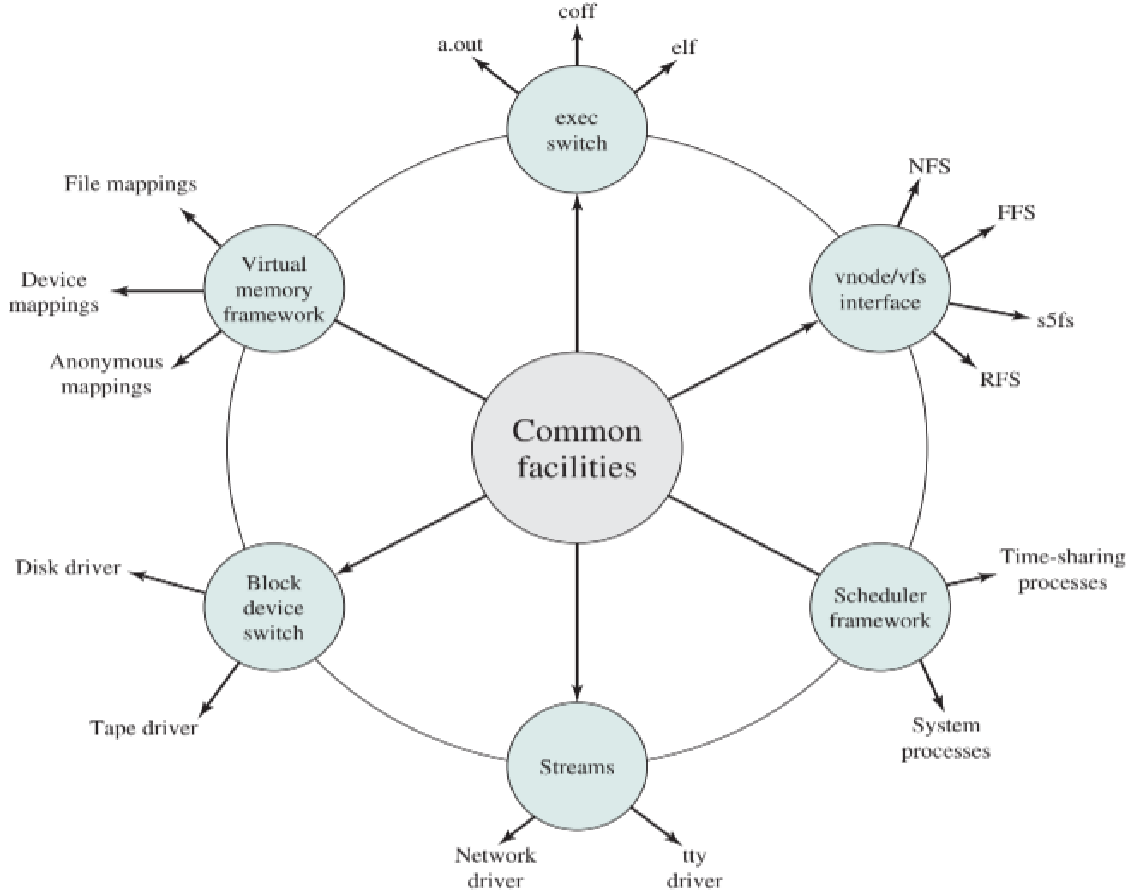


Figure 2.18 **Modern UNIX Kernel**

As UNIX evolved, the number of different implementations proliferated, each providing some useful features. There was a need to produce a new implementation that unified many of the important innovations, added other modern OS design features, and produced a more modular architecture. Typical of the modern UNIX kernel is the architecture depicted in Figure.

Some examples of modern UNIX systems.

System V Release 4 (SVR4)

SVR4, developed jointly by AT&T and Sun Microsystems, combines features from SVR3, 4.3BSD, Microsoft Xenix System V, and SunOS. It was almost a total 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.**

BSD

The Berkeley Software Distribution (BSD) series of UNIX releases have played a key role in the development of OS design theory. One of the most widely used and best documented versions of BSD is FreeBSD. FreeBSD is **popular** for **Internet-based servers and firewalls and is used in a number of embedded systems.** The latest version of the Macintosh OS, Mac OS X, is based on FreeBSD 5.0 and the Mach 3.0 microkernel.

Solaris 10

Solaris is Sun’s SVR4-based UNIX release, with the latest version being 10. Solaris provides **all of the features of SVR4 plus a number of more advanced features, such as a fully preemptable, multithreaded kernel, full support for SMP, and an object-oriented interface to file systems. Solaris is the most widely used and most successful commercial UNIX implementation.**

LINUX

History

Linux started out as a UNIX variant for the IBM PC (Intel 80386) architecture. Linus Torvalds, a Finnish student of computer science, wrote the initial version.

Torvalds posted an early version of Linux on the Internet in 1991. Since then, a number of people, collaborating over the Internet, have contributed to the development of Linux, all under the control of Torvalds.

Because Linux is free and the source code is available, it became an early alternative to other UNIX workstations, such as those offered by Sun Microsystems and IBM. Today, Linux is a full-featured UNIX system that runs on all of these platforms and more, including Intel Pentium and Itanium, and the Motorola/IBM PowerPC.

Modular Structure

Most UNIX kernels are monolithic. 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.**

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.

• **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.

The principal kernel components are
Signals
System calls
Processes and scheduler
Virtual memory
File systems
Network protocols
Traps and faults
Physical memory
Interrupts

LINUX VSERVER VIRTUAL MACHINE ARCHITECTURE

Linux VServer is an open-source, fast, lightweight approach to implementing virtual machines on a Linux server. Only a single copy of the Linux kernel is involved. VServer consists of a relatively modest modification to the kernel plus a small set of OS userland tools.

(The term userland refers to all application software that runs in user space rather than kernel space.)

The VServer Linux kernel supports a number of separate virtual servers. The kernel manages all system resources and tasks, including process scheduling, memory, disk space, and processor time.

Each virtual server is isolated from the others using Linux kernel capabilities.

This provides security and makes it easy to set up multiple virtual machines on a single platform.

Processes: Process description and Control

WHAT IS A PROCESS?

Various Definitions

- 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 the execution of a sequence of instructions, a current state, and an associated set of system resources.

PROCESS STATES

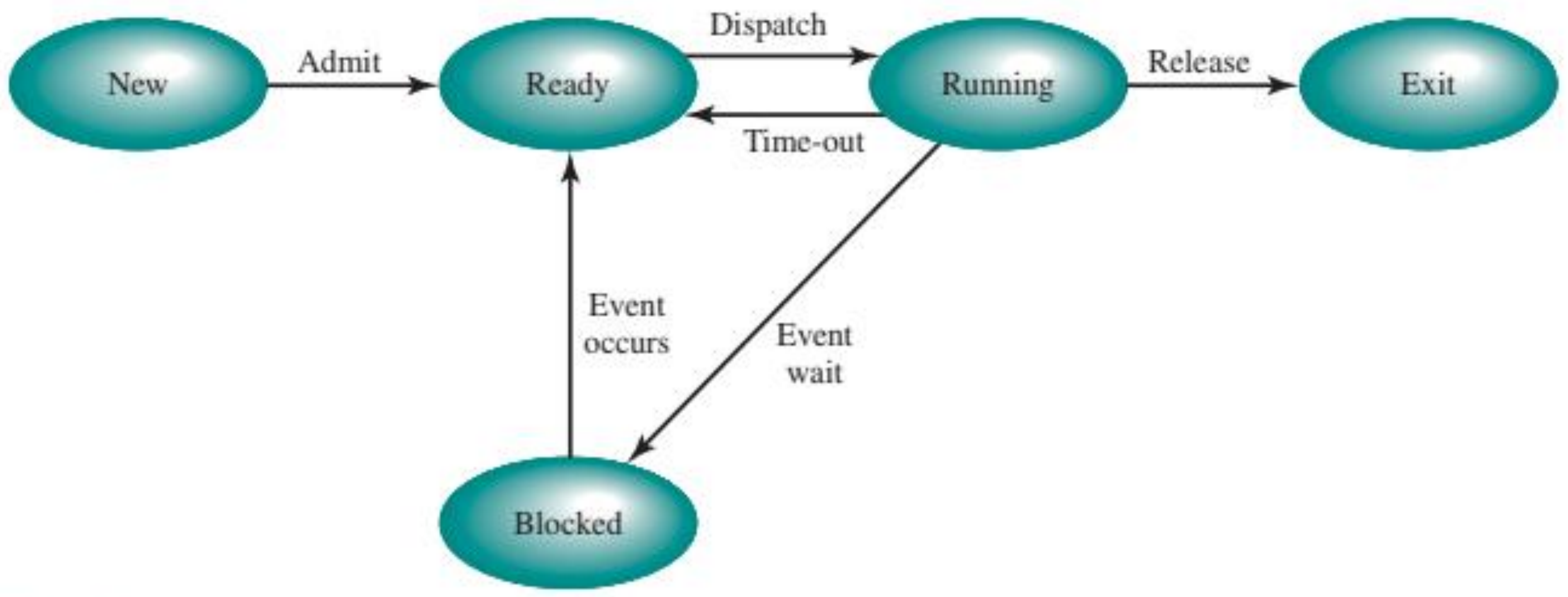


Figure 3.6 **Five-State Process Model**

- **Running:** The process that is currently being executed. For this chapter, we will assume a computer with a single processor, so at most one process at a time can be in this state.

- **Ready:** A process that is prepared to execute when given the opportunity.

- **Blocked/Waiting:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.

- **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.

- **Exit:** A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.

PROCESS DESCRIPTION

Operating System Control Structures

The OS **controls** events within the computer system. It **schedules** and **dispatches** processes for execution by the processor, **allocates resources** to processes, and **responds to requests by user** processes for basic services. Fundamentally, we can think of the OS as that entity that **manages the use of system resources by processes**.

If the OS is **to manage processes and resources, it must have information about the current status of each process and resource.** The OS **constructs and maintains tables of information about each entity that it is managing**. A general idea of the scope of this effort is indicated in Figure , which shows **four different types of tables maintained by the OS**: **memory, I/O, file, and process**. Although the details will differ from one OS to another, fundamentally, all operating systems maintain information in these four categories.

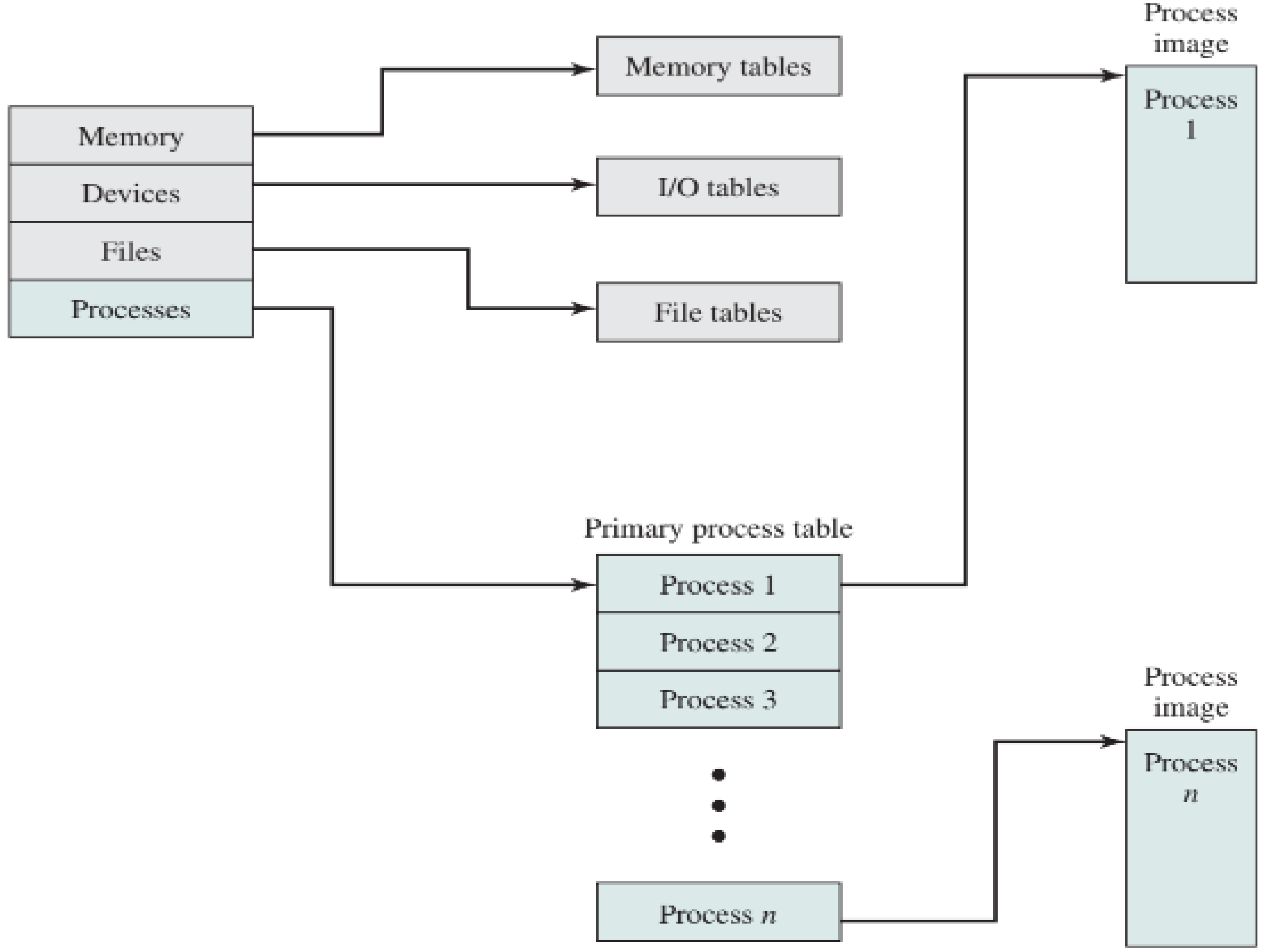


Figure 3.11 **General Structure of Operating System Control Tables**

Memory tables are used to keep track of both main (real) and secondary (virtual) memory.
The memory tables must include the following information:

- The **allocation of main memory** to processes
- The **allocation of secondary memory** to processes
- Any **protection attributes** of blocks of main or virtual memory, such as which processes may access certain shared memory regions
- Any **information needed to manage virtual memory**

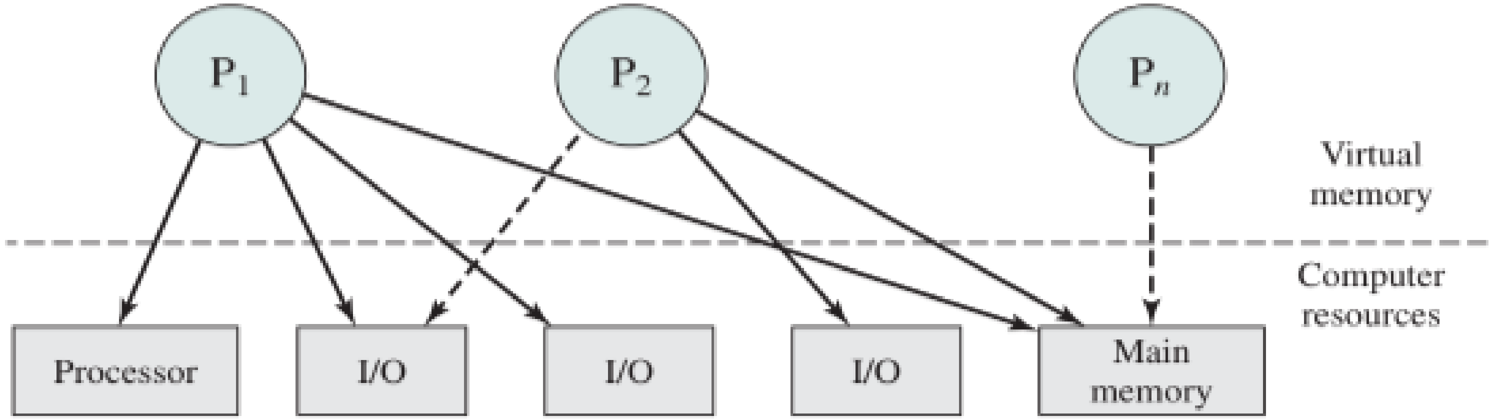


Figure 3.10 **Processes and Resources (resource allocation at one snapshot in time)**

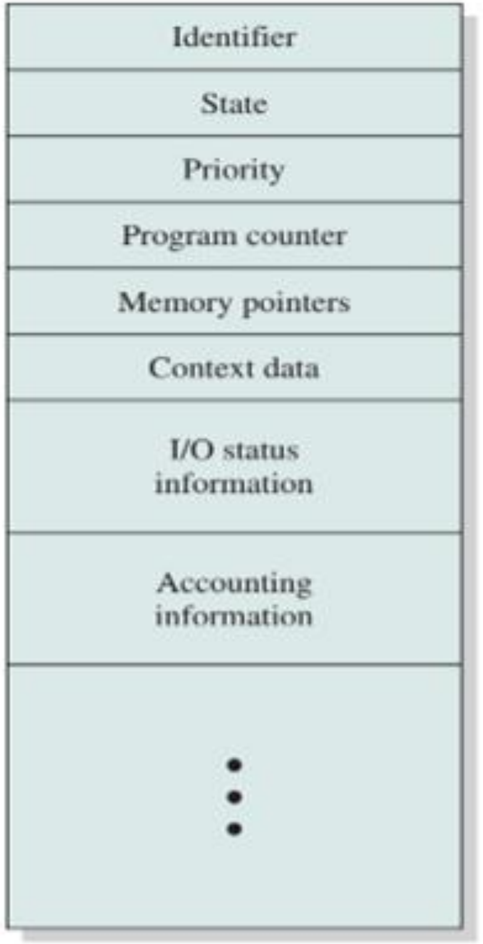
I/O tables are used by the OS to manage the I/O devices and channels of the computer system. At any given time, an I/O device may be available or assigned to a particular process. If an I/O operation is in progress, the OS needs to know the status of the I/O operation and the location in main memory being used as the source or destination of the I/O transfer.

The OS may also maintain **file table**s . **These tables provide information about the existence of files, their location on secondary memory, their current status, and other attributes.**

The OS must maintain process tables to manage processes. Process table contains **Process Control Block.**

Processes: Process description and Control

At any given point in time,
while the program is executing , the process can be uniquely characterized by a number of elements, including the following:



e 3.1 Simplified Process Control Block

- **Identifier:** A unique identifier associated with this process, to distinguish it from all other processes.
- **State:** If the process is currently executing, it is in the running state.
- **Priority:** Priority level relative to other processes.
- **Program counter:** The address of the next instruction in the program to be executed.
- **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:** These are data that are present in registers in the processor while the process is executing.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., disk drives) assigned to this process, a list of files in use by the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

PROCESS CONTROL

Modes of Execution

Most processors support at least two modes of execution. Certain instructions can only be executed in the more-privileged mode. **The more-privileged** mode is referred to as the system mode, control mode, or kernel mode.

The **less-privileged mode** is often referred to as the user mode, because user programs typically would execute in this mode.

Kernel mode encompasses the important system functions like process creation and termination, process scheduling and dispatching, process switching, allocation of address space to processes, swapping, allocation of I/O channels and devices to processes, Interrupt handling etc.

Process Creation

1. Assign a unique process identifier to the new process.
2. Allocate space for the process.
3. Initialize the process control block.
4. Set the appropriate linkages. E.g. Put in ready queue
5. Create or expand other data structures.

Process Switching

At some time, a running process is interrupted and the OS assigns another process to the Running state and turns control over to that process. Process switching is done in various situations Clock interrupt, I/O interrupt, Memory fault etc.

Mode Switching

Change Of Process State