Mount Kenya University

DIGITAL VARSITY,

SCHOOL OF COMPUTING AND INFORMATICS

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: BIT2104

COURSE TITLE: OPERATING SYSTEMS

Course Description

The course introduces the basic principles of operating systems in which the student will be introduced to the role of the operating systems in controlling and coordinating all the operations of a computer. This module introduces the learner to operating system structures, processes and services, process management aspects and scheduling. The course also introduces the concepts of memory management, mass storage structure, Input/output structure and computer security features. The module summarizes each chapter with a case study and review questions to evaluate learning outcomes against course objectives.

Expected Learning Outcomes:

At the end of the course, the student should be able to:

- Understand the concept of an operating system through the historical developments.
- Understand the role of operating systems in computer systems.
- Apply concepts of operating systems process, synchronization and communication to practical problems.
- Understand resource management in computer systems and the subsequent application in memory, file and disk management.
- Understand the underlying principles of operating system design.

Course Structure

Evolution of operating system, Types of operating systems, desirable characteristics of modern operating systems, Functions of operating systems: multiprogramming, resource allocation and management and their implementation, supervisory services, memory management and data management services. Process management: process and program concepts, process coordination and synchronization, process scheduling, inter process communication, real time clock management. Deadlock: deadlock condition, causes of deadlocks, detection and prevention of deadlock. Memory management: types of memory, objectives of memory management, memory allocation schemes, virtual. File management: objectives of file management, file concepts, types of files, file organization, file systems, file access, directories. I/O allocation; device drivers

Contact hours: 42

Pre-requisites: BIT 1102: Computer Applications
BIT 1103: Computer Architecture

Course Outline
WEEK 1 & WEEK 2

CHAPTER ONE: INTRODUCTION TO OPERATING SYSTEMS

- Introduction to Operating System, Functions of Operating System
- Operating System as User Interface
- I/O System Management
- Assembler, Compiler, Loader

WEEK3
CHAPTER TWO: Evolution of Operating Systems

- Earliest Computers
- Von Neumann architecture
- Bare Hardware, Monitors, Microprocessors
- System Calls
- Device drivers and library functions

WEEK 4

CHAPTER THREE: Operating-System Modes and Operations

- Operating System Modes
- Operating System Operations
- Batch System, Time Sharing System
- Multiprogramming
- Spooling

WEEK 5 & WEEK6

CHAPTER FOUR: Operating System's Process Management

- Processes and Programs, Process State
- Suspended Processes
- Process Control Block
- Process Management
- Scheduling Queues, Process Synchronization
- Threads and Deadlocks

WEEK 7 & WEEK8

CHAPTER FIVE: Memory Management

- Introduction
- Memory Partitioning
- Swapping
- Paging
- Segmentation

WEEK 9

CHAPTER SIX: File Management

- File Concept, File Support, Access Methods
- Directory Systems
- File Protection
- Free Space Management

WEEK 10 &WEEK11

CHAPTER SEVEN: Input Output Hardware

- Principal of I/O Hardware
- Polling, I/O Devices
- Device Controllers
- Direct Memory Access

WEEK 12

CHAPTER EIGHT: INPUT/OUTPUT SOFTWARE

- Principle of I/O Software , Application I/O Interfaced ,Interrupts
- Clocks and Timers
- Blocking and Non-blocking I/O
- Kernel I/O Subsystem ,Scheduling
- Buffering ,Caching
- Spooling and Device Reservation ,Error Handling
- Device Drivers

Course Assessment

Examination	- 70%
Continuous Assessment Test (CATS)	- 20%
Assignments	- 10%
Total	- 100%

Core Reading Materials:

- Silberschatz, A. and Galvin B.: *Operating System Concepts* 7th Edition, Wiley Higher Education, 2007

Recommended Reading materials:

- Stallings, W.: *Operating Systems: Internals and Design Principles* 5th Edition**,** Addison-Wesley, 2004
- Tanenbaum, A. and Woodhull, A. *Operating Systems* 2nd Edition, Prentice Hall, 1997.
- Silberschatz, A. and Galvin B. *Operating System Concepts* 5th Edition, Addison Wesley, 1999.
- Davis, William S.: *Operating Systems* 5th Edition, Addison Wesley, 2001.
- Abrahams, Paul W.: *UNIX for the Impatient* Addison Wesley, 1992.

TABLE OF CONTENTS

Contents

Chapter 1
1.0 Introduction to Operating Systems
Unit Structure

- Objectives
- Introduction
- Operating System
- Definition of operating system
- Functions of Operating System
- Operating System as User Interface
- I/O System Management
- Assembler
- Compiler
- Loader
- Summary
- Review Questions

Objectives

After going through this unit, you will be able to:

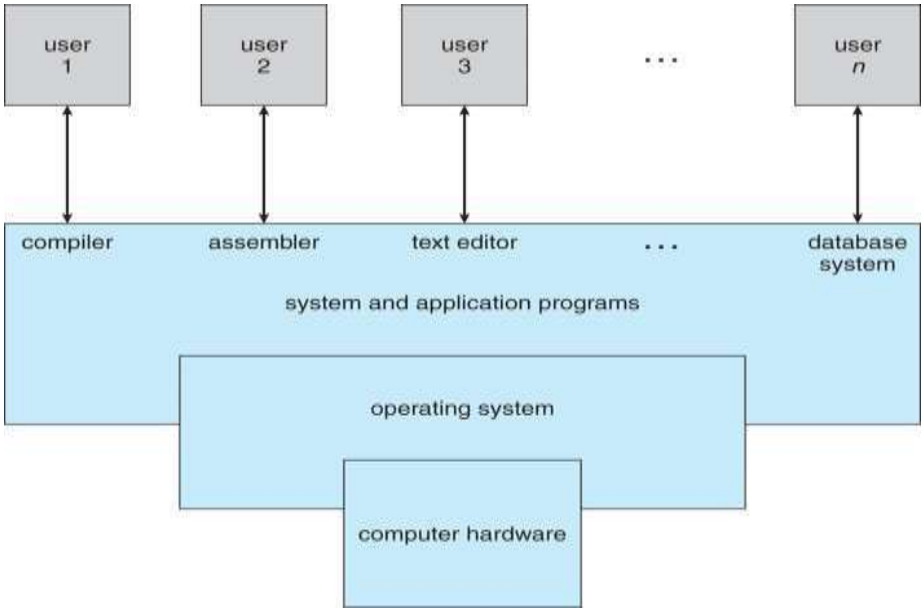
- Describe Basic Organization of Computer Systems
- Define Operating system, functions, and Relationship between Software and Hardware
- Define assembler, linker, loader, compiler
- Explain basic I/O Relationships

Introduction to Operating Systems

An ***operating system*** is a program that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. An amazing aspect of operating systems is how varied they are in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for handheld computers are designed to provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be ***convenient***, others to be ***efficient***, and others some combination of the two.

1.01What Operating Systems Do

We begin our discussion by looking at the operating system's role in the overall computer system. A computer system can be divided roughly into four components: the ***hardware***, the



operating system, the ***application programs***, and the ***users*** (Figure 1).

Figure 1

The ***hardware***—the ***central processing unit (CPU),*** the ***memory***, and the ***input/output (I/O) devices***—provides the basic computing resources for the system. The ***application programs***—such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users. We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these
resources in the operation of the computer system. An operating system is similar to a ***government.*** Like a government, it performs no useful function by itself. It simply provides an ***environment*** within which other programs can do useful work.

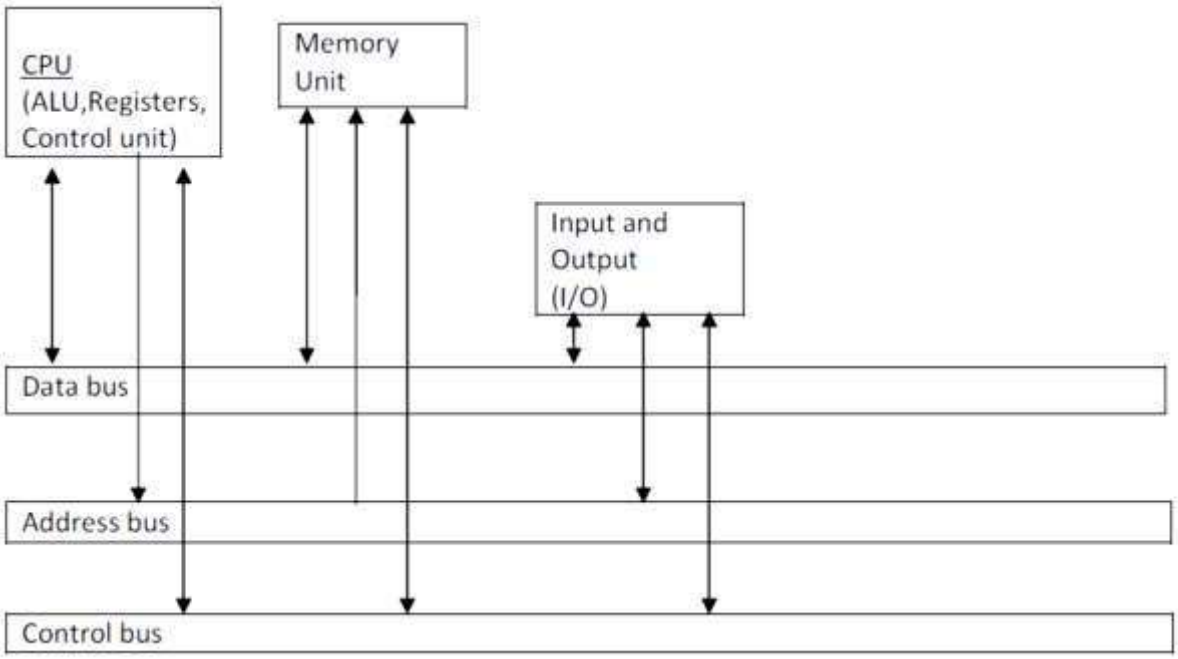


Figure 2

1.02 Registers

Are special purpose, high –speed temporary memory units, they hold various types of information such as data, instructions, addresses, and the intermediate results of calculations. Essentials they hold the information that the CPU is currently working on. They are as the CPU's working memory. As soon as a particular instruction or piece of data is processed, the next instruction immediately replaces it, and the information that that results from the processing is returned to main memory (RAM). Data and instructions do not enter either the ALU or the CU, instead the ALU works on the data held in the registers along with the instructions on which it acts. Instruction addresses are normally stored in consecutive registers and are executed sequentially. The control unit reads an instruction in the memory by a specific address in the register and executes it. The next instruction is then fetched from the sequence and executed. This type of instruction sequencing is possible through the use of a counter which calculates the address of the instruction after instruction. This counter is a register, which stores intermediate data used during the execution of the instructions after it is read from the memory. Program counter (pc)-keeps track of next instruction to be executed.

1.03 Control unit

It is the heart of the CPU. It controls the I/O devices and transfer of data to and from the primary storage. Instructions are retrieved from the primary storage, one at a time .For this, the control unit uses the instruction register for holding the current instruction, and instruction pointer to hold the address of the next instruction.

Note:

The control unit therefore, controls how other parts of the CPU and in turn, rest of the computer system should work in the order that the instructions are executed in correct manner. In order to maintain the proper sequence of events required for any processing task, the control unit uses clock inputs.

1.04 Basic Operations of the control unit (CU)

a) Fetching

Is the processing of obtaining a program instructions or data item from memory.

b) Decoding

Is the processing of translating instruction into commands the computer can execute.

c) Executing

It is he the process of carrying out the commands.

d) Storing:

It is the process of writing the results to the memory.

1.05 Relationship between Software and Hardware

It's the blending of software and hardware that gives life to a computer system. The software and the hardware share a special relationship whereby the hardware share a special relationship whereby the hardware is the heart and the software is its soul. Both are complimentary to each other.

Software categories

1) System software:

It provides the basic non-task-specific functions of the computer.

2) Application software:

It is utilized by users to accomplish specific tasks. System software is the software that is essential for the computer to function.

1.06 System Software

It is the software that contributes to the control and performance of the computer system and permits the user to use the system more conveniently. System software comprises programs written in low-level languages which interact with the hardware at a very basic level. They are the basic necessity of the computer system for its proper functioning. Note: System software not only controls the hardware but also provides a platform for other programs to run onto them. System software can be further divided into two categories:

a) System management programs.

b) System development programs.

a) System management programs

They are responsible for the management and accurate functioning of the computer system. It includes an integrated system of programs that manage the operations of the processor, control the I/O, manage storage resources, and provide various support services as the computer executes application programs. Examples include:

- ✓ Operating system

- ✓ Device drivers
- ✓ System utilities

1.07 Operating System Roles Summary

An operating system is responsible for performing basic tasks such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the hard disk, and controlling peripheral devices such as printers and modems. It also ensures that different programs executes at the same time do not interfere with each other. It provides a software platform on top of which other programs can run. In general –Operating system organizes and controls the hardware. Basic Operations of Operating System:

- ✓ Memory management- As a memory manager, the operating system handles allocation and deallocation of memory space as required by various programs.
- ✓ Process management- As a process manager, the operating system handles the creation and deletion of processes, suspension and resumption of processes, and also scheduling and synchronization of processes.
- ✓ File management- Operating system is responsible for creation and deletion of files and directories. It also takes care of other file related activities such as organizing, storing, and protecting the files.
- ✓ Device management-Operating system provides I/O subsystem between processes and device driver. It handles the device caches, buffers, and interrupts. Operating system also detects device failure and notifies the same to the user.
- ✓ Security management-Operating system protects system resources and information against destruction and unauthorized use.
- ✓ User interface-operating system provides the interface between the user and the hardware. The user interface is the layer that actually interacts with the computer operator.

a) Device Drivers

They are system programs, which are responsible for proper functioning of devices. Every device or hardware e.g printer, monitor, mouse or keyboard has a driver program for support. Whenever a new device is added to the computer system, a new device driver must be installed before the driver can be used. A driver acts like a translator between the device and programs that use the device. For example, when a user prints a document, the processor issues a set of generic commands to the printer driver, and the driver translates those commands into specialized instructions that the printer understands.

b) System Utilities

They perform day-to-day tasks related to the maintenance of the computer system. They are used to:

Support, enhance ,expand and secure existing programs and data in the computer system.

1.08 Functions of System Utilities

i) Back up- Sometimes data files can get corrupted, or get accidentally deleted. In such case,

data back-ups become very useful.

ii) Virus protection- Antivirus programs are essential system software for a system functioning

in a network. Viruses are small programs written with malicious intent, which copy themselves to the local system hardware drives from other infected systems. Virus keeps on spreading to the computers through the network.

iii) Disk Management-They include various system software like defragmenting disks, data compression software and formatting disks tools.

De-fragmentation is putting fragments of files in a sequential order onto the disk. This reduces the time to access the files. Data compression programs squeeze out the stack space generated by the formatting schemes. Formatting tools format the hard drive in tracks and sectors for orderly storing of the data in the drive.

1.09 System development programs

They consist of system software which are associated with the development of computer programs. Program development tools allow programmer to write and construct programs that the operating system can execute. Software is developed to accomplish a particular task. Certain tools are required to build software and they include:

- ✓ Appropriate computer language
- ✓ Translator to translate a particular language to machine language.

Examples of system development programs are:

- ❖ Programming languages
- ❖ Language translators
- ❖ Linkers and loaders

i) **Programming languages**

A programming language is a primary interface of a programmer with. A program is an ordered list of instructions that, when executed, causes the computer to behave in a predetermined manner. A programming language includes a series of commands. Examples of programming languages include -Machine language- is a computer language composed of 0s and 1s, which is directly executable by the computer.

ii) **Assembly language-** is a programming language which that consists of a group of coded letters or labels, called mnemonics. Assembler accepts instructions written in assembly language of

the computer and translates them into a binary representation of the corresponding machine instructions.

iii) **High level language**-is a programming language that is closer to the English language in which each instruction is equivalent of many machine language instructions.

iv) **Language translators**

Is a tool which is used to translate a program written in a programming language to machine language. Translators accept the programs written in a programming language and executes them by transforming them into a suitable form for execution, i.e , conversion of programming

language to machine language-(0s and 1s) that the computer is able to process. Examples include:

- ✓ **Compiler**- As a system program, the compiler translates source code (user written program) into object code (binary form). The compiler looks at the entire piece of source code and reorganizes the instructions.
- ✓ **Interpreter**- It analyses and executes the source code in-line-by-line manner, without looking at the entire program, that is, an interpreter translates a statement in a program and executes the statement immediately before translating the next source language statement.

v) **Linkers and loaders**

A linker is system program that links together several object modules and libraries to form a single, coherent, program (executable). The loader is responsible for loading and relocation of the executable program in the main memory. The functions of a loader include assigning load time space for storage, that is, storage allocation and to assist the program to execute appropriately.

1.10 Summary

An Operating system is concerned with the allocation of resources and services, such as memory, processors, devices and information. The Operating System correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

Assembler

Input to an assembler is an assembly language program. Output is an object program plus information that enables the loader to prepare the object program for execution.

Loader

A loader is a routine that loads an object program and prepares it for execution. There are various loading schemes: absolute, relocating and direct-linking. In general, the loader must load, relocate, and link the object program

Compilers

A compiler is a program that accepts a source program in a high-level language and produces a corresponding object program.

1.11 Review Questions

Q. 1 Define Operating System?
Q. 2 Explain various function of operating system?
Q. 3 Explain I/O system Management?
Q. 4 Define & explain Assembler, Loader, Compiler?

Chapter 2
2.01 Evolution of Operating Systems
Unit Structure

- Earliest Computers
- Von Neumann architecture
- Bare Hardware
- Monitors
- Microprocessors
- System Calls
- Device drivers and library functions
- Summary
- Review Question

Objectives

After going through this unit, you will be able to:

- Describe Basic Organization of Computer Systems
- Define Operating system, functions, history and Evolution
- Define system calls, Device Drivers and Library functions
- Introduce Unix and Microprocessor technologies

Evolution of Operating Systems

2.02 Earliest computers
The earliest calculating machine was the abacus, believed to have been invented in Babylon around 2400 B.C.E. The abacus was used by many different cultures and civilizations, including the major advance known as the Chinese abacus from the 2nd Century B.C.E.The Chinese developed the South Pointing Chariot in 115 B.C.E. This device featured a differential gear, later used in modern times to make analog computers in the mid-20th Century.

The Indian grammarian Panini wrote the *Ashtadhyayi* in the 5th Century B.C.E. In this work he created 3,959 rules of grammar for India's Sanskrit language. This important work is the oldest surviving linguistic book and introduced the idea of metarules, transformations, and recursions, all of which have important applications in computer science.

The first true computers were made with intricate gear systems by the Greeks. These computers turned out to be too delicate for the technological capabilities of the time and were abandoned as impractical. The Antikythera mechanism, discovered in a shipwreck in 1900, is an early mechanical analog computer from between 150 B.C.E. and 100 B.C.E.. The Antikythera mechanism used a system of 37 gears to compute the positions of the sun and the moon through the zodiac on the Egyptian calendar, and possibly also the fixed stars and five planets known in antiquity (Mercury, Venus, Mars, Jupiter, and Saturn) for any time in the future or past. The

system of gears added and subtracted angular velocities to compute differentials. The Antikythera mechanism could accurately predict eclipses and could draw up accurate astrological charts for important leaders. It is likely that the Antikythera mechanism was based on an astrological computer created by Archimedes of Syracuse in the 3rd century B.C.E.

The first digital computers were made by the Inca using ropes and pulleys. Knots in the ropes served the purpose of binary digits. The Inca had several of these computers and used them for tax and government records. In addition to keeping track of taxes, the Inca computers held data bases on all of the resources of the Inca empire, allowing for efficient allocation of resources in response to local disasters (storms, drought, earthquakes, etc.). Spanish soldiers acting on orders of Roman Catholic priests destroyed all but one of the Inca computers in the mistaken belief that any device that could give accurate information about distant conditions must be a divination device powered by the Christian "Devil" (and many modern Luddites continue to view computers as satanically possessed devices).

In the 1800s, the first computers were programmable devices for controlling the weaving machines in the factories of the Industrial Revolution. Created by Charles Babbage, these early computers used Punch cards as data storage (the cards contained the control codes for the various patterns). These cards were very similiar to the famous Hollerinth cards developed later. The first computer programmer was Lady Ada, for whom the Ada programming language is named.

In 1822 Charles Babbage proposed a difference engine for automated calculating. In 1933 Babbage started work on his Analytical Engine, a mechanical computer with all of the elements of a modern computer, including control, arithmetic, and memory, but the technology of the day couldn't produce gears with enough precision or reliability to make his computer possible. The Analytical Engine would have been programmed with Jacquard's punched cards. Babbage designed the Difference Engine No.2. Lady Ada Lovelace wrote a program for the Analytical Engine that would have correctly calculated a sequence of Bernoulli numbers, but was never able to test her program because the machine wasn't built.

George Boole introduced what is now called Boolean algebra in 1854. This branch of mathematics was essential for creating the complex circuits in modern electronic digital computers.

In the 1900s, researchers started experimenting with both analog and digital computers using vacuum tubes. Some of the most successful early computers were analog computers, capable of performing advanced calculus problems rather quickly. But the real future of computing was digital rather than analog. Building on the technology and math used for telephone and telegraph switching networks, researchers started building the first electronic digital computers.

The first modern computer was the German Zuse computer (Z3) in 1941. In 1944 Howard Aiken of Harvard University created the Harvard Mark I and Mark II. The Mark I was primarily mechanical, while the Mark II was primarily based on reed relays. Telephone and telegraph

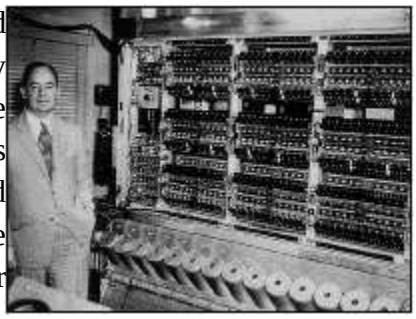
companies had been using reed relays for the logic circuits needed for large scale switching networks.The first modern electronic computer was the ENIAC in 1946, using 18,000 vacuum tubes. See below for information on Von Neumann's important contributions. The first solid-state (or transistor) computer was the TRADIC, built at Bell Laboratories in 1954. The transistor had previously been invented at Bell Labs in 1948.

2.03 Von Neumann architecture

John Louis von Neumann, mathematician (born János von Neumann 28 December 1903 in Budapest, Hungary, died 8 February 1957 in Washington, D.C.), proposed the *stored program concept* while professor of mathemtics (one of the orginal six) at Princeton University's Institute for Advanced Services, in which programs (code) are stored in the same memory as data. The computer knows the difference between code and data by which it is attempting to access at any given moment. When evaluating code, the binary numbers are decoded by some kind of physical logic circuits (later other methods, such as microprogramming, were introduced), and then the instructions are run in hardware. This design is called von Neumann architecture and has been used in almost every digital computer ever made.Von Neumann architecture introduced flexibility to computers. Previous computers had their programming hard wired into the computer. A particular computer could only do one task (at the time, mostly building artillery tables) and had to be physically rewired to do any new task. By using numeric codes, von Neumann computers could be reprogrammed for a wide variety of problems, with the decode logic remaining the same.

As processors (especially super computers) get ever faster, the *von Neumann bottleneck* is starting to become an issue. With data and code both being accessed over the same circuit lines, the processor has to wait for one while the other is being fetched (or written). Well designed data and code caches help, but only when the requested access is already loaded into cache. Some researchers are now experimenting with Harvard architecture to solve the von Neumann bottleneck. In Harvard architecture, named for Howard Aiken's experimental Harvard Mark I (ASCC) calculator [computer] at Harvard University, a second set of data and address lines along with a second set of memory are set aside for executable code, removing part of the conflict with memory accesses for data.

Von Neumann became an American citizen in 1933 to be eligible to help on top secret work during World War II. There is a story that Oskar Morganstern coached von Neumann and Kurt Gödel on the U.S. Constitution and American history while driving them to their immigration interview. Morganstern asked if they had any questions, and Gödel replied that he had no questions, but had found some logical inconsistencies in the Constitution that he wanted to ask the Immigration officers about. Morganstern recommended that he not ask questions, but just answer them.Von Neumann occasionally worked with Alan Turing in 1936 through 1938 when Turing was a graduate student at Princeton. Von Neumann was exposed to the concepts of logical design and universal machine proposed in Turing's 1934 paper "On Computable Numbers with an Application to the Entschiedungs-problem".



Von Neumann worked with such early computers as the Harvard Mark I, ENIAC, EDVAC, and his own IAS computer. Early research into computers involved doing the computations to create tables, especially artillery firing tables. Von Neumann was convinced that the future of computers involved applied mathematics to solve specific problems rather than mere table generation. Von Neumann was the first person to use computers for mathematical physics and economics, proving the utility of a general purpose computer.

Von Neumann proposed the concept of stored programs in the 1945 paper "First Draft of a Report on the EDVAC". Influenced by the idea, Maurice Wilkes of the Cambridge University Mathematical Laboratory designed and built the EDSAC, the world's first operational, production, stored-program computer.

The *first stored computer program* ran on the Manchester Mark I [computer] on June 21, 1948.

Von Neumann foresaw the advantages of parallelism in computers, but because of construction limitations of the time, he worked on sequential systems.Von Neumann advocated the adoption of the bit as the measurement of computer memory and solved many of the problems regarding obtaining reliable answers from unreliable computer components.Interestingly, von Neumann was opposed to the idea of compilers. When shown the idea for FORTRAN in 1954, von Neumann asked "Why would you want more than machine language?". Von Neumann had graduate students hand assemble programs into binary code for the IAS machine. Donald Gillies, a student at Princeton, created an assembler to do the work. Von Neumann was angry, claiming "It is a waste of a valuable scientific computing instrument to use it to do clerical work".

Von Neumann also did important work in set theory (including measure theory), the mathematical foundation for quantum theory (including statistical mechanics), self-adjoint algebras of bounded linear operators on a Hilbert space closed in weak operator topology, non-linear partial differential equations, and automata theory (later applied to computers). His work in economics included his 1937 paper "A Model of General Economic Equilibrium" on a multi-sectoral growth model and his 1944 book "Theory of Games and Economic Behavior" (co-authored with Morgenstern) on game theory and uncertainty.

2.04 Bare hardware

In the earliest days of electronic digital computing, everything was done on the bare hardware. Very few computers existed and those that did exist were experimental in nature. The researchers who were making the first computers were also the programmers and the users. They worked directly on the "bare hardware". There was no operating system. The experimenters wrote their programs in machine or assembly language and a running program had complete control of the entire computer. Often programs and data were entered by hand through the use of toggle

switches. Memory locations (both data and programs) could be read by viewing a series of lights (one for each binary digit). Debugging consisted of a combination of fixing both the software and hardware, rewriting the object code and changing the actual computer itself.

The lack of any operating system meant that only one person could use a computer at a time. Even in the research lab, there were many researchers competing for limited computing time. The first solution was a reservation system, with researchers signing up for specific time slots. The earliest billing systems charged for the entire computer and all of its resources (regardless of whether used or not) and was based on outside clock time, being billed from the scheduled start to scheduled end times.

The high cost of early computers meant that it was essential that the rare computers be used as efficiently as possible. The reservation system was not particularly efficient. If a researcher finished work early, the computer sat idle until the next time slot. If the researcher's time ran out, the researcher might have to pack up his or her work in an incomplete state at an awkward moment to make room for the next researcher. Even when things were going well, a lot of the time the computer actually sat idle while the researcher studied the results (or studied memory of a crashed program to figure out what went wrong). Simply loading the programs and data took up some of the scheduled time.

2.05 Computer operators

One solution to this problem was to have programmers prepare their work off-line on some input medium (often on punched cards, paper tape, or magnetic tape) and then hand the work to a computer operator. The computer operator would load up jobs in the order received (with priority overrides based on politics and other factors). Each job still ran one at a time with complete control of the computer, but as soon as a job finished, the operator would transfer the results to some output medium (punched tape, paper tape, magnetic tape, or printed paper) and deliver the results to the appropriate programmer. If the program ran to completion, the result would be some end data. If the program crashed, memory would be transferred to some output medium for the programmer to study (because some of the early business computing systems used magnetic core memory, these became known as "core dumps").The concept of computer operators dominated the mainframe era and continues today in large scale operations with large numbers of servers.

2.06 Device drivers and library functions

Soon after the first successes with digital computer experiments, computers moved out of the lab and into practical use. The first practical application of these experimental digital computers was the generation of artillery tables for the British and American armies. Much of the early research in computers was paid for by the British and American militaries. Business and scientific applications followed. As computer use increased, programmers noticed that they were duplicating the same efforts.

Every programmer was writing his or her own routines for I/O, such as reading input from a magnetic tape or writing output to a line printer. It made sense to write a common device driver for each input or output device and then have every programmer share the same device drivers rather than each programmer writing his or her own. Some programmers resisted the use of common device drivers in the belief that they could write “more efficient” or faster or "“better” device drivers of their own.

Additionally each programmer was writing his or her own routines for fairly common and repeated functionality, such as mathematics or string functions. Again, it made sense to share the work instead of everyone repeatedly “reinventing the wheel”. These shared functions would be organized into libraries and could be inserted into programs as needed. In the spirit of cooperation among early researchers, these library functions were published and distributed for free, an early example of the power of the open source approach to software development.

Computer manufacturers started to ship a standard library of device drivers and utility routines with their computers. These libraries were often called a runtime library because programs connected up to the routines in the library at run time (while the program was running) rather than being compiled as part of the program. The commercialization of code libraries ended the widespread free sharing of software. Manufacturers were pressured to add security to their I/O libraries in order to prevent tampering or loss of data.

2.07 Input output control systems

The first programs directly controlled all of the computer’s resources, including input and output devices. Each individual program had to include code to control and operate each and every input and/or output device used. One of the first consolidations was placing common input/output (I/O) routines into a common library that could be shared by all programmers. I/O was separated from processing.

These first rudimentary operating systems were called an Input Output Control System or IOCS. Computers remained single user devices, with main memory divided into an IOCS and a user section. The user section consisted of program, data, and unused memory. The user remained responsible for both set up and tear down. Set up included loading data and program, by front panel switches, punched card, magnetic tapes, paper tapes, disk packs, drum drives, and other early I/O and storage devices. Paper might be loaded into printers, blank cards into card punch machines, and blank or formatted tape into tape drives, or other output devices readied.

Tear down would include un-mounting tapes, drives, and other media. The very expensive early computers sat idle during both set up and tear down. This waste led to the introduction of less expensive I/O computers. While one I/O computer was being set up or torn down, another I/O computer could be communicating a readied job with the main computer. Some installations might have several different I/O computers connected to a single main computer to keep the expensive main computer in use. This led to the concept of multiple I/O channels.

2.08 Monitors

As computers spread from the research labs and military uses into the business world, the accountants wanted to keep more accurate counts of time than mere wall clock time.

This led to the concept of the monitor. Routines were added to record the start and end times of work using computer clock time. Routines were added to I/O library to keep track of which devices were used and for how long. With the development of the Input Output Control System, this time keeping routines were centralized. You will notice that the word monitor appears in the name of some operating systems, such as FORTRAN Monitor System. Even decades later many programmers still refer to the operating system as the monitor. An important motivation for the creation of a monitor was more accurate billing. The monitor could keep track of actual use of I/O devices and record runtime rather than clock time. For accurate time keeping the monitor had to keep track of when a program stopped running, regardless of whether it was a normal end of the program or some kind of abnormal termination (such as a crash).The monitor reported the end of a program run or error conditions to a computer operator, who could load the next job waiting, rerun a job, or take other actions. The monitor also notified the computer operator of the need to load or unload various I/O devices (such as changing tapes, loading paper into the printer, etc.).

1950s

Some operating systems from the 1950s include: FORTRAN Monitor System, General Motors Operating System, Input Output System, SAGE, and SOS.SAGE (Semi-Automatic Ground Environment), designed to monitor weapons systems, was the first real time control system.

2.09 Batch systems

Batch systems automated the early approach of having human operators load one program at a time. Instead of having a human operator load each program, software handled the scheduling of jobs. In addition to programmers submitting their jobs, end users could submit requests to run specific programs with specific data sets (usually stored in files or on cards). The operating system would schedule “batches” of related jobs. Output (punched cards, magnetic tapes, printed material, etc.) would be returned to each user.

General Motors Operating System, created by General Motors Research Laboratories in early 1956 (or late 1955) for thieir IBM 701 mainframe is generally considered to be the first batch operating system and possibly the first “real” operating system. The operating system would read in a program and its data, run that program to completion (including outputting data), and then load the next program in series as long as there were additional jobs available.

Batch operating systems used a Job Control Language (JCL) to give the operating system instructions. These instructions included designation of which punched cards were data and which were programs, indications of which compiler to use, which centralized utilities were to be run, which I/O devices might be used, estimates of expected run time, and other details. This

type of batch operating system was known as a single stream batch processing system. Examples of operating systems that were primarily batch-oriented include: BKY, BOS/360, BPS/360, CAL, and Chios.

Early 1960s

The early 1960s saw the introduction of time sharing and multi-processing. Some operating systems from the early 1960s include: Admiral, B1, B2, B3, B4, Basic Executive System, BOS/360, Compatible Timesharing System (CTSS), EXEC I, EXEC II, Honeywell Executive System, IBM 1410/1710 OS, IBSYS, Input Output Control System, Master Control Program, and SABRE. The first major transaction processing system was SABRE (Semi-Automatic Business Related Environment), developed by IBM and American Airlines.

2.10 Multiprogramming

There is a huge difference in speed between I/O and running programs. In a single stream system, the processor remains idle for much of the time as it waits for the I/O device to be ready to send or receive the next piece of data. The obvious solution was to load up multiple programs and their data and switch back and forth between programs or jobs. When one job idled to wait for input or output, the operating system could automatically switch to another job that was ready.

System calls

The first operating system to introduce system calls was University of Machester's Atlas I Supervisor.

Time sharing

The operating system could have additional reasons to rotate through jobs, including giving higher or lower priority to various jobs (and therefore a larger or smaller share of time and other resources). The Compatible Timesharing System (CTSS), first demonstrated in 1961, was one of the first attempts at timesharing.

While most of the CTSS operating system was written in assembly language (all previous OSes were written in assembly for efficiency), the scheduler was written in the programming language MAD in order to allow safe and reliable experimentation with different scheduling algorithms. About half of the command programs for CTSS were also written in MAD.Timesharing is a more advanced version of multiprogramming that gives many users the illusion that they each have complete control of the computer to themselves. The scheduler stops running programs based on a slice of time, moves on to the next program, and eventually returns back to the beginning of the list of programs. In little increments, each program gets their work done in a manner that appears to be simultaneous to the end users.

Mid 1960s

Some operating systems from the mid-1960s include: Atlas I Supervisor, DOS/360, Input Output Selector, Master Control Program, and Multics.The Atlas I Supervisor introduced spooling, interrupts, and virtual memory paging (16 pages) in 1962. Segmentation was introduced on the Burroughs B5000. MIT's Multics combined paging and segmentation.The Compatible Timesharing System (CTSS) introduced email.

Late 1960s

Some operating systems from the late-1960s include: BPS/360, CAL, CHIPPEWA, EXEC 3, EXEC 4, EXEC 8, GECOS III, George 1, George 2, George 3, George 4, IDASYS, MASTER, Master Control Program, OS/MFT, OS/MFT-II, OS/MVT, OS/PCP, and RCA DOS.

2.11 Microprocessors

In 1968 a group of scientists and engineers from Mitre Corporation (Bedford, Massachusetts) created Viatron Computer company and an intelligent data terminal using an 8-bit LSI microprocessor from PMOS technology. A year later in 1969 Viatron created the 2140, the first 4-bit LSI microprocessor. At the time MOS was used only for a small number of calculators and there simply wasn't enough worldwide manufacturing capacity to build these computers in quantity.Other companies saw the benefit of MOS, starting with Intel's 1971 release of the 4-bit 4004 as the first commercially available microprocessor. In 1972 Rockwell released the PPS-4 microprocessor, Fairchild released the PPS-25 microprocessor, and Intel released the 8-bit 8008 microprocessor. In 1973 National released the IMP microprocessor. In 1973 Intel released the faster NMOS 8080 8-bit microprocessor, the first in a long series of microprocessors that led to the current Pentium.

In 1974 Motorola released the 6800, which included two accumulators, index registers, and memory-mapped I/O. Monolithic Memories introduced bit-slice micro processing. In 1975 Texas Instruments introduced a 4-bit slice microprocessor and Fairchild introduced the F-8 microprocessor.

Early 1970s

Some operating systems from the early-1970s include: BKY, Chios, DOS/VS, Master Control Program, OS/VS1, and UNIX.In 1970 Ken Thompson of AT&T Bell Labs suggested the name "Unix" for the operating system that had been under development since 1969. The name was an intentional pun on AT&T's earlier Multics project (*uni-* means "one", *multi-* means "many").

2.12 UNIX takes over mainframes

UNIX was orginally developed in a laboratory at AT&T's Bell Labs (now an independent corporation known as Lucent Technologies). At the time, AT&T was prohibited from selling computers or software, but was allowed to develop its own software and computers for internal use. A few newly hired engineers were unable to get valuable mainframe computer time because

of lack of seniority and resorted to writing their own operating system (UNIX) and programming language (C) to run on an unused mini-computer .

The computer game Space Travel was originally written by Jeremy Ben for Multics. When AT&T pulled out of the Multics project, J. Ben ported the program to FORTRAN running on GECOS on the GE 635. J. Ben and Dennis Ritchie ported the game in DEC PDP-7 assembly language. The process of porting the game to the PDP-7 computer was the beginning of Unix.Unix was originally called UNICS, for Uniplexed Information and Computing Service, a play on words variation of Multics, Multiplexed Information and Computing Service.

AT&T's consent decree with the U.S. Justice Department on monopoly charges was interpreted as allowing AT&T to release UNIX as an open source operating system for academic use. Ken Thompson, one of the originators of UNIX, took UNIX to the University of California, Berkeley, where students quickly started making improvements and modifications, leading to the world famous Berkeley Standard Distribution (BSD) form of UNIX.

Vendors such as Sun, IBM, DEC, SCO, and HP modified Unix to differentiate their products. This splintered Unix to a degree, though not quite as much as is usually perceived. Necessity being the mother of invention, programmers has created development tools that help them work around the differences between Unix flavors. As a result, there is a large body of software based on source code that will automatically configure itself to compile on most Unix platforms, including Intel-based Unix. Regardless, Microsoft would leverage the perception that Unix is splintered beyond hope, and present Windows NT as a more consistent multi-platform alternative." —Nicholas Petreley, "The new Unix alters NT's orbit", NC World.

2.13 UNIX to the Desktop

Among the early commercial attempts to deploy UNIX† on desktop computers was AT&T selling UNIX in an Olivetti box running a w74 680x0 assembly language is discussed in the assembly language section. Microsoft partnered with Xenix to sell their own version of UNIX.[w74] Apple computers offered their A/UX version of UNIX running on Macintoshes. None of these early commercial UNIXs was successful. "Unix started out too big and unfriendly for the PC. … It sold like ice cubes in the Arctic. … Wintel emerged as the only 'safe' business choice", Nicholas Petreley.[w74].

"Unix had a limited PC market, almost entirely server-centric. SCO made money on Unix, some of it even from Microsoft. (Microsoft owns 11 percent of SCO, but Microsoft got the better deal in the long run, as it collected money on each unit of SCO Unix sold, due to a bit of code in SCO Unix that made SCO somewhat compatible with Xenix. The arrangement ended in 1997.)" —Nicholas Petreley, "The new Unix alters NT's orbit", NC World.

Mid 1970s

In 1973 the kernel of Unix was rewritten in the C programming language. This made Unix the world's first portable operating system, capable of being easily ported (moved) to any hardware. This was a major advantage for Unix and led to its widespread use in the multi-platform environments of colleges and universities.

Late 1970s

Some operating systems from the late-1970s include: EMAS 2900, General Comprehensive OS, VMS (later renamed OpenVMS), OS/MVS.

1980s

Some operating systems from the 1980s include: AmigaOS, DOS/VSE, HP-UX, Macintosh, MS-DOS, and ULTRIX. The 1980s saw the commercial release of the graphic user interface, most famously the Apple Macintosh, Commodore Amiga, and Atari ST, followed by Microsoft's Windows.

1990s

Some operating systems from the 1990s include: BeOS, BSDi, FreeBSD, NeXT, OS/2, Windows 95, Windows 98, and Windows NT.

2000s

Some operating systems from the 2000s include: Mac OS X, Syllable, Windows 2000, Windows Server 2003, Windows ME, and Windows XP

Generation	Year	Electronic devices used	Types of OS and devices
First	1945 – 55	Vacuum tubes	Plug boards
Second	1955 – 1965	Transistors	Batch system
Third	1965 – 1980	Integrated Circuit (IC)	Multiprogramming
Fourth	Since 1980	Large scale integration	PC

2.14 Summary

An operating system provides services to programs and to the users of those programs. It provided by one environment for the execution of programs. The services provided by one operating system is difficult than other operating system. Operating system makes the programming task easier.

Batch operating system is one where programs and data are collected together in a batch before processing starts. In batch operating system memory is usually divided into two areas: Operating system and user program area.

Time sharing, or multitasking, is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running. When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system.

2.15 Review Questions

Q.1Outline the evolution of Operating Systems across computer generations

Q.2 What are the key features of Multiprogramming?

Q.4Distinguish between batch systems and Time sharing in computing ?

Q.3 Explain Basic architecture of UNIX/Linux system?

Q.4 Explain basic features of UNIX/Linux?

Chapter 3

3.0 Operating-System Modes and Operations

Unit Structure

- Objectives
- Operating System Modes
- Operating System Operations
- Batch System
- Time Sharing System
- Multiprogramming
- Spooling
- Summary
- Review Questions
- Objectives

Objective

After going through this unit, you will be able to:

- To describe the services an operating system provides to users, processes, modes and other systems
- Describe operating system services and its components.
- Define multitasking and multiprogramming.
- Describe timesharing, buffering & spooling.
-

Operating-System Modes and Operations

From our discussion about computer-system organization and architecture, we are ready to talk about operating systems. An operating system provides the environment within which programs are executed. Internally, operating systems vary greatly in their makeup, since they are organized along many different lines. There are, however, many commonalities, which we consider in this section.

One of the most important aspects of operating systems is the ability to multiprogramming. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute. The idea is as follows: The operating system keeps several jobs in memory simultaneously (Figure 3). Since, in general, main memory is too small to accommodate all jobs; the jobs are kept initially on the disk in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory.

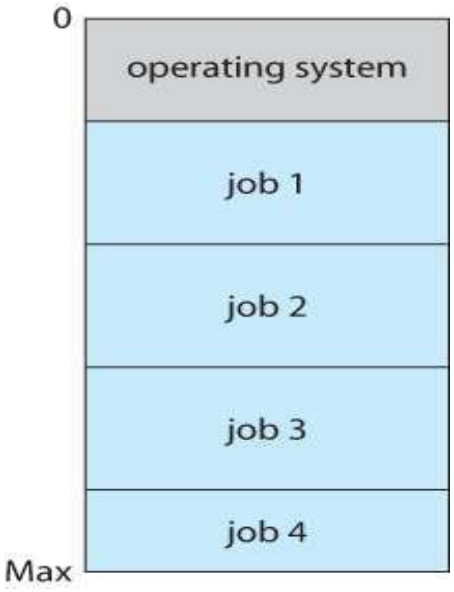


Figure 3 Memory layout for a multiprogramming system

The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multi-programmed system, the CPU would sit idle. In a multi-programmed system, the operating system simply switches to, and executes, another job. When that job needs to wait, the CPU is switched to another job, and so on. Eventually, the first job finishes waiting and gets the
CPU back. As long as at least one job needs to execute, the CPU is never idle. This idea is common in other life situations. A lawyer does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.) Multi-programmed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system. ***Time sharing*** (or ***multitasking***) is a logical extension of ***Multi-programming***. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. Time sharing requires an interactive (or hands-on) computer system, which provides direct communication between the user and the system. The ser gives instructions to the operating system or to a program directly, using a input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the response time should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multi-programming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output goes to a display for the user, and input comes from a user keyboard, mouse, or other device. Since interactive I/O

typically runs at "people speeds," it may take a long time to complete. Input, for example, may be bounded by the user's typing speed; seven characters per second is fast for people but incredibly slow for computers. Rather than let the CPU sit idle as this interactive input takes place, the operating system will rapidly switch the CPU to the program of some other user. Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management, which is covered in later chapters.. In addition, if several jobs are ready to run at the same time, the system must choose among them. Making this decision is CPU scheduling, which is also discussed later. Finally, running multiple jobs concurrently requires that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management.

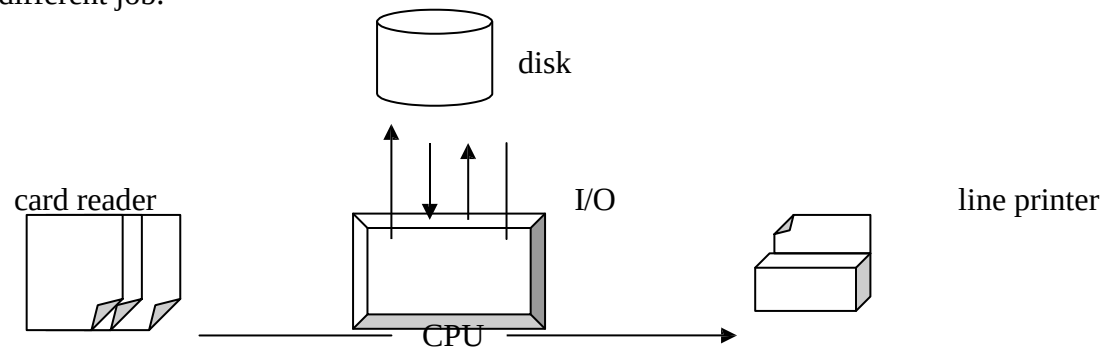
3.01 Simple Batch Systems

Here the user prepares a job (which consists of program, data and some control information), submits it to the computer operator and receives the output after the program is executed. The job is usually punched on cards and the output is usually printed. To speed up processing, jobs with similar needs can be batched together and run as a group. Thus, the operator sorts the programs with similar requirements into batches, runs each batch and sends the output to the programmer. Therefore in a batch operating system there is a lack of interaction between the user and the job while that job is executing.

The operating system is always in memory and its job is to transfer control automatically from one job to the next. As the I/O devices are slower than the speed of the CPU (the difference may be three orders of magnitude or more), the CPU is often idle. For using the system more efficiently, cards can be read from the card reader onto the disk. The operating system records their location on disk in a table. When a job requests the printer, the output is copied into a system buffer and is written to the disk. When the job is completed, the output is printed. This form of processing is called spooling (simultaneous peripheral operation on line). Spooling is also used for processing data at remote sites. The CPU send the data via communication paths to

a remote site and the processing is done with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Spooling overlaps the I/O of one job with the computation of other jobs. During the execution of one job, the spooler may be reading the input of another job while printing the output of a different job.



3.02 Multi-programmed Batch Systems

Spooling provides a job pool on disk. In order to increase CPU utilization, jobs in the pool may be scheduled in different ways, such as, first-come first-served, shortest job first, priority basis, etc. The most important aspect of job scheduling is the ability to multiprogramming. The operating system keeps several jobs (a subset of jobs in the job pool) in memory at a time. During the execution of one job, if it waits for an I/O operation to complete, the operating system switches to and executes another job. In this way, as long as there is always some job to execute, the CPU will never be idle.

3.03 Time-Sharing Systems

In multi-programmed batch systems, the user cannot interact with the program during its execution. In time-sharing or multitasking systems, multiple jobs are executed by the CPU switching between them, but the switching occurs so frequently that the user may interact with each program while it is running. Time-sharing operating systems:

- ✓ uses CPU scheduling and multiprogramming,
- ✓ uses time-slice mechanism,
- ✓ allows interactive I/O,
- ✓ allows many users to share the computer simultaneously.

3.04 Personal Computer (PC) Systems

A computer system dedicated to a single user is referred to as a PC. In the first PCs, the operating system was neither multiuser nor multitasking (eg. MS-DOS). The operating system concepts used in mainframes and minicomputers, today, are also used in PCs (eg. UNIX, Microsoft Windows NT, Macintosh OS).

3.05 Parallel Systems

Parallel systems have more than one processor. In multiprocessor systems (tightly coupled), processors are in close communication, such as they share computer bus, clock, memory or peripherals.

3.06 Distributed Systems

Distributed systems also have more than one processor. Each processor has its local memory. Processors communicate through communication lines (eg. Telephone lines, high-speed bus, etc.). Processors are referred to as sites, nodes, computers depending on the context in which they are mentioned. Multicomputer systems are loosely coupled. Example applications are e-mail, web server, etc.

3.07 Real-time Systems

Real-time systems are special purpose operating systems. They are used when there are rigid time requirements on the operation of a processor or the flow of data, and thus it is often used as a control device in a dedicated application (eg. fuel injection systems, weapon systems, industrial control systems, …). It has well defined, fixed time constraints. The processing must be done within the defined constraints, or the system fails. Two types:

Hard real-time systems guarantee that critical tasks complete on time.

In Soft real-time systems, a critical real-time task gets priority over other tasks, and the task retains that priority until it completes.

3.08 Operating-System Modes

As mentioned earlier, modern operating systems are interrupt driven. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an ***interrupt*** or a ***trap***. A trap (or an ***exception)*** is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. ***An interrupt service routine*** is provided that is responsible for dealing with the interrupt. Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running. With sharing, many processes could be adversely affected by a bug in one program.

For example, if a process gets stuck in an infinite loop, this loop could prevent the correct operation of many other processes. More subtle errors can occur in a multiprogramming system, where one erroneous program might modify another program, the data of another program, or even the operating system itself.

Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

3.09 Dual-Mode Operation

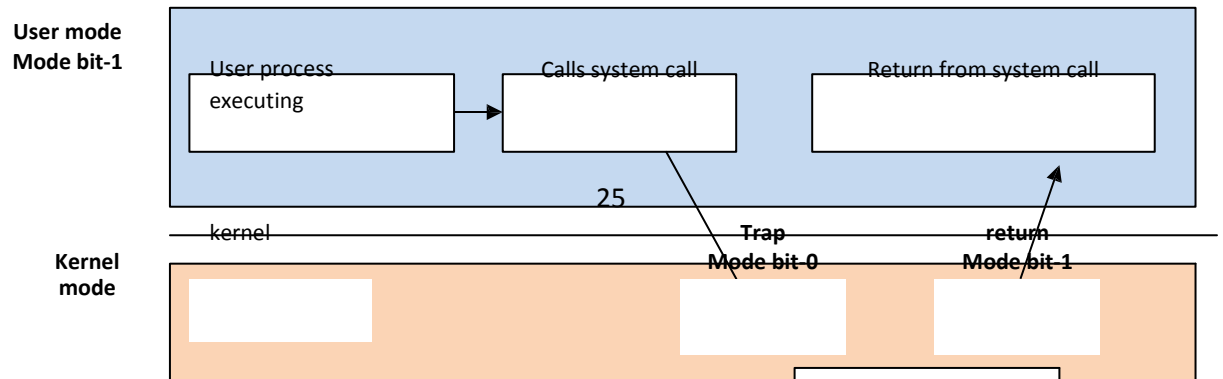
In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user-defined code. The approach taken by most computer systems is to provide hardware support that allows us to differentiate among various modes of execution.At the very least, we need two separate modes of operation: ***user mode*** and ***kernel mode*** (also called ***supervisor mode***, ***system mode***, or ***privileged mode***). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1).With the mode bit, we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), it must transition from user to kernel mode to fulfill the request. This is shown in Figure 1.3. As we shall see, this architectural enhancement is useful for many other aspects of system operation as well. At system boot time, the hardware starts in kernel mode. The ***operating system*** is then loaded and starts user applications in ser mode.

Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as privileged instructions. The hardware allows privileged

Instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system. The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management, and interrupt management. As we shall see throughout the text, there are many additional privileged instructions.

We can now see the life cycle of instruction execution in a computer system. Initial control resides in the operating system, where instructions are executed in kernel mode.



When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call. System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, depending on the functionality provided by the underlying processor. In all forms, it is them method used by a process to request action by the operating system.

3.10 Summary

The operating system must ensure correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, the hardware has two modes: ***user mode and kernel mode***. Various instructions (such as I/O instructions and halt instructions) are privileged and can be executed only in kernel mode. The memory in which the operating system resides must also be protected from modification by the user. A timer prevents infinite loops. These facilities (dual mode, privileged instructions, memory protection, and timer interrupt) are basic building blocks used by operating systems to achieve correct operation.
A process (or job) is the fundamental unit of work in an operating system.

Process management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with each other.
An operating system manages memory by keeping track of what parts of memory are being used and by whom. The operating system is also responsible for dynamically allocating and freeing memory space. Storage space is also managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.
Operating systems must also be concerned with protecting and securing the operating system and users. Protection measures are mechanisms that control the access of processes or users to the resources made available by the computer system. Security measures are responsible for defending a computer system from external or internal attacks.
Distributed systems allow users to share resources on geographically dispersed hosts connected via a computer network. Services may be provided through either the ***client –server model*** or the peer-to-peer model. In a clustered system, multiple machines can perform computations on data

residing on shared storage, and computing can continue even when some subset of cluster members fails.

3.11 Review Questions

Q. 1 Explain various operating system services?

Q. 2 Define Spooling? Describe Spooling process?

Q. 3 Differentiate user mode & Kernel Mode?

Q. 4 explain the circumstances that occasions interrupt and interrupt service routines ?

Chapter 4

4.0 Operating System's Process Management

- Objectives
- Concept of Process
- Processes and Programs
- Process State
- Suspended Processes
- Process Control Block
- Process Management
- Scheduling Queues
- Threads
- Process Synchronization
- Deadlocks
- Summary
- Review Questions

Objectives

After going through this unit, you will be able to:

- To introduce the notion of a process – a program in execution, which forms the basis of all computation
- Define threads and their roles in computing systems
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explain deadlocks, their causes and solutions

A ***process*** can be thought of as a program in execution. A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task. These resources are allocated to the process
either when it is created or while it is executing. A process is the unit of work in most systems. Systems consist of a collection of processes: Operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently. Although traditionally a process contained only a single ***thread*** of control as it ran, most modern operating systems now support processes that have multiple threads.
The operating system is responsible for the following activities in connection with process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

Early computer systems allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, current-day computer systems allow multiple programs to be loaded into memory and executed concurrently.

This evolution required firmer control and more compartmentalization of the various programs, and these needs resulted in the notion of a process, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating system

processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, you

will read about what processes are and how they work.

4.01 The Process

A process is more than the program code, which is sometimes known as the ***text section***. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's reisters. A process generally also includes the process ***stack***, which contains temporary data (such as function

parameters, return addresses, and local variables), and a ***data section***, which contains global variables. A process may also include ***a heap***, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure 5.

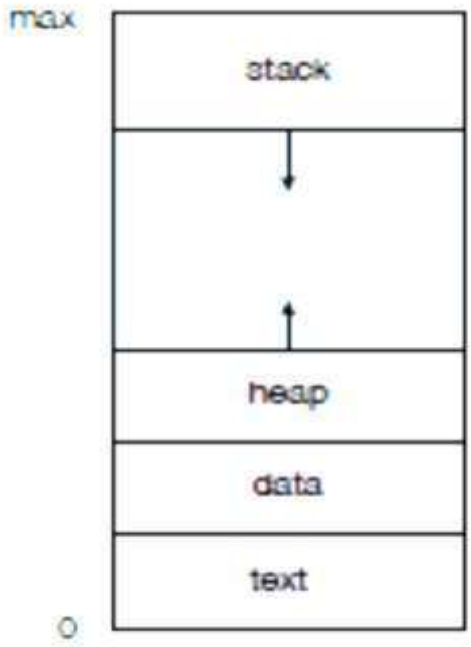


Figure 5 Process in memory

We emphasize that a program by itself is not a process; a program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file),whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out.)

Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the Web browser program. Each of these is a separate process, and although the text sections are equivalent, the data, heap, and stack sections vary.

4.02 Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states:

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

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also delineate process states more

finely. It is important to realize that only one process can be ***running*** on any processor at any instant. Many processes may be ***ready*** and ***waiting***, however. The state diagram corresponding to these states is presented in Figure 5.

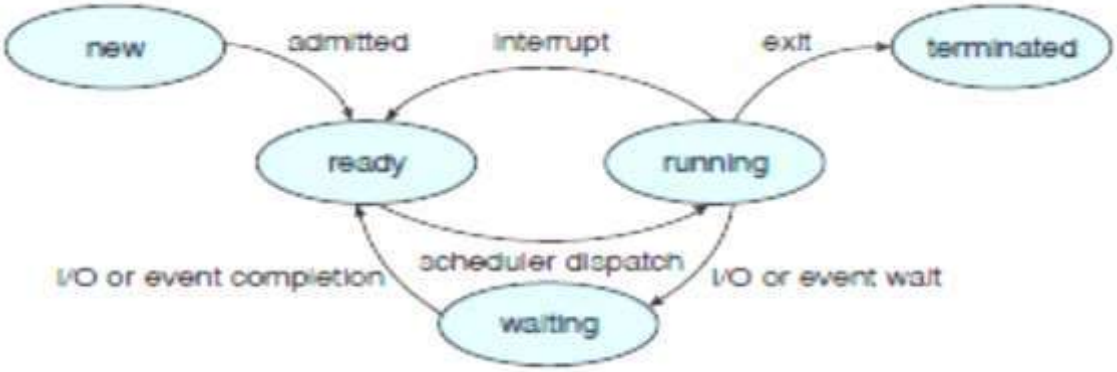


Figure 5 Process State

4.03 Process Control Block

Each process is represented in the operating system by a **process control block** (**PCB**)—also called ***a task control block***. A ***PCB*** is shown in Figure 3.3. It contains many pieces of information associated with a specific process, including ***these***:

Process state Process number Program counter
Registers
Memory limits List of open files
…

Figure 6 Process Control Block

- ✓ **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- ✓ **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- ✓ **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-

purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 6).

- ✓ **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- ✓ **Memory-management information**. This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system .
- ✓ **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- ✓ **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

4.04 Threads

A ***thread*** is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or ***heavyweight)*** process as a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 7 illustrates the difference between a traditional ***single-threaded*** process and a ***multithreaded*** process text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background. In certain situations, a single application may be required to perform several similar tasks. For example, a Web server accepts client requests for Web pages, images, sound, and so forth. A busy Web server may have several (perhaps thousands of) clients concurrently accessing it. If the Web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads.

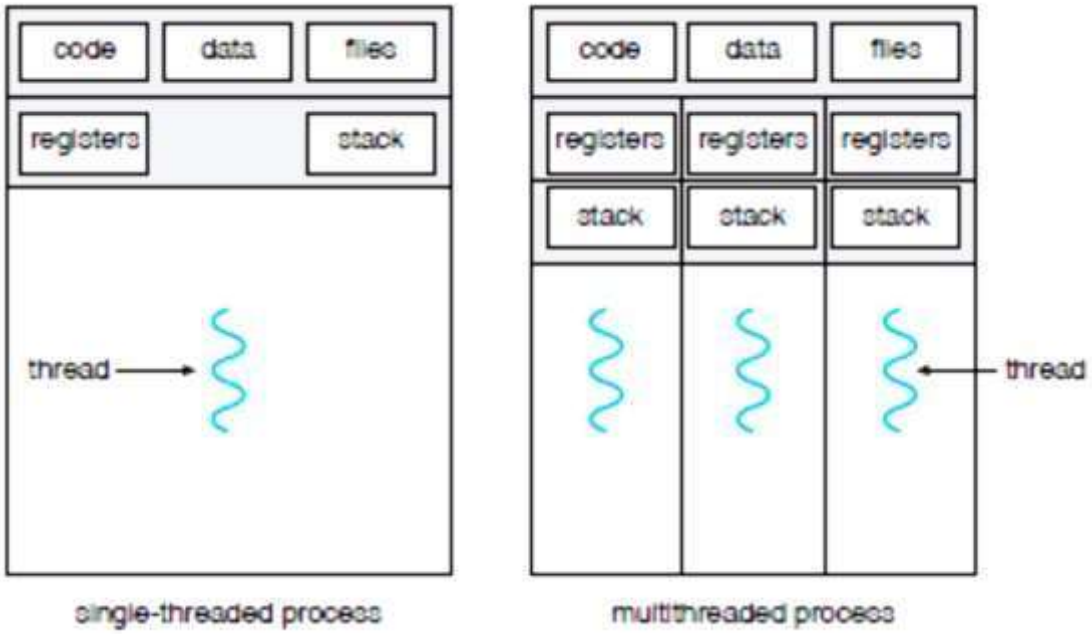


Figure 7 Single thread and multithread process

4.05 CPU Scheduling

CPU scheduling is the basis of multi-programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms. We also consider the problem of selecting an algorithm for a particular system.

On operating systems that support threads, it is kernel-level threads—not processes—that are in fact being scheduled by the operating system. However, the terms **process scheduling** and **thread scheduling** are often used interchangeably. We use *process scheduling* when discussing general scheduling concepts and *thread scheduling* to refer to thread-specific ideas.

In a single-processor system, only one process can run at a time; any others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this

time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that

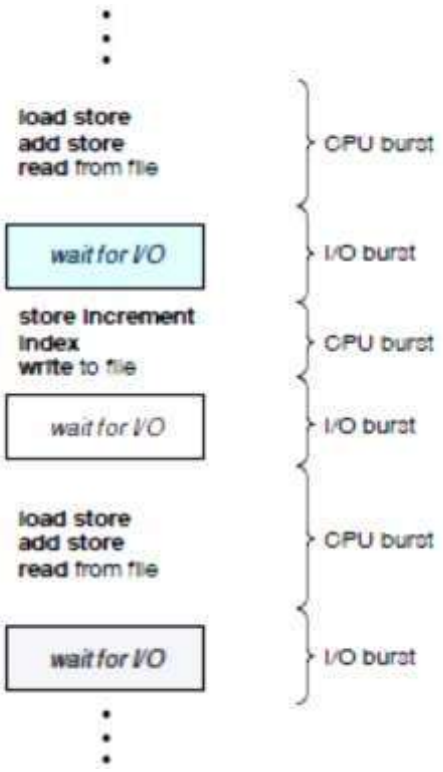


Figure 8 Alternating sequences of CPU and I/O bursts

process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

4.06 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler** (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a **first-in, first-out (FIFO)** queue. As we shall see when we consider the various **scheduling algorithms**, a ready queue can be implemented as a **FIFO** queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

4.07 Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1) When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)
2) When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3) When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4) When a process terminates for situations 1 and 4, there is no choice in terms of scheduling. A new process(if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **non-preemptive** or **cooperative**; otherwise, it is **preemptive**. Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x; Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used ***preemptive scheduling***.

4.08 Process Synchronization

A cooperating process is one that can affect or be affected by other processes executing in a system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads, earlier discussed. Concurrent access to shared data may result in data inconsistency, however. There are various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

4.09 The Critical-Section Problem

Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a ***critical section***, in which the process may be changing common variables, updating a table, writing a file, and so on.

The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The *critical-section problem* is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the ***entry section***. The critical section may be followed by an ***exit section***. The remaining code is the ***remainder section***. The general structure of a typical process *Pi* is shown below

do *{ entry*
section critical
section *exit*
section
remainder section
} while (TRUE);

General structure of a typical process *Pi* .

The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

i) **Mutual exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

ii) **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

iii) **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the ***relative speed*** of the *n* processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (*kernel code*) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems: (1) **preemptive kernels** and (2) **non-preemptive kernels**. A pre-emptive kernel allows a process to be preempted while it is running in kernel mode. A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. Obviously, a non-preemptive kernel is essentially free from race conditions on kernel data structures, as only

one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

4.10 Classic Problems of Synchronization

In this section, we present a number of synchronization problems as examples of a large class of on currency-control problems. These problems are used for testing nearly every newly proposed

synchronization scheme. In our solutions to the problems, we use semaphores for synchronization.

4.11 The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously,
chaos may ensue.
To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the *readers–writers problem*. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers–writers problem has several variations, all involving priorities. The simplest one, referred to as the *first* readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The *second* readers –writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization.

4.12 The Bounded-Buffer Problem

We assume that the pool consists of *n* buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0. The code for the producer process is and the code for the consumer process is shown below. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do {
wait(full);
wait(mutex);
```

. . .
// remove an item from buffer to nextc
. . .
signal(mutex);
signal(empty);
. . .
// consume the item in nextc
. . .
} while (TRUE);

The structure of the consumer process.

do *{*
. . .
// produce an item in nextp
. . .
wait(empty);
wait(mutex);
. . .
// add nextp to buffer
. . .
signal(mutex);
signal(full);
} while (TRUE);

The structure of the producer process

4.13 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging

do *{*
wait(mutex);
readcount++;
if (readcount == 1)
wait(wrt);
signal(mutex);
. . .
// reading is performed
. . .
wait(mutex);
readcount--;
if (readcount == 0)
signal(wrt);
signal(mutex);
} while (TRUE);

The structure of a reader process.

4.14 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur. Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- ✓ A process might access a resource without first gaining access permission to the resource.
- ✓ A process might never release a resource once it has been granted access to the resource.
- ✓ A process might attempt to release a resource that it never requested.
- ✓ A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about
the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.
One possible solution to the current problem is to include the resource access operations within the **ResourceAllocator** monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.
To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the ResourceAllocator monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

4.15 Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**. Deadlocks can occur with many types of resources available in a computer system.
A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. **Release**. The process releases the resource.

The request and release of resources are system calls

4.16 Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. **Mutual exclusion**. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait**. A set *{P*0, *P*1, ..., *Pn}* of waiting processes must exist such that *P*0 is waiting for a resource held by *P*1, *P*1 is waiting for a resource held by *P*2, ..., *Pn*−1 is waiting for a resource held by *Pn*, and *Pn* is waiting for a resource held by *P*0.

All four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

4.17 Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

i) We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
ii) We can allow the system to enter a deadlocked state, detect it, and recover.
iii) We can ignore the problem altogether and pretend that deadlocks never occur in the system.
iv) The third solution is the one used by most operating systems, including UNIX andWindows; it is then up to the application developer to write programs that handle deadlocks.

Next, we elaborate briefly on each of the three methods for handling deadlocks. Before proceeding, we should mention that some researchers have argued that none of the basic approaches alone is appropriate for the entire spectrum of resource-allocation problems in

operating systems. The basic approaches can be combined, however, allowing us to select an optimal approach for each class of resources in a system.

To ensure that deadlocks never occur, the system can use either a ***deadlock prevention*** or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. If a system does not employ either a deadlock-prevention or a ***deadlock avoidance algorithm***, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred).

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlock state yet has no way of recognizing what has happened. In this case, the undetected deadlock will result in deterioration of the system's performance, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems.

4.18 Summary

A process is a program in execution. As a **process** executes, it changes state. The state of a process is defined by that process's current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated.

Each process is represented in the operating system by its own **process control block (PCB).**

A process, when it is not executing, placed in some waiting queue. There are two major classes of queues in an operating system: I/O request queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB and the PCBs can be linked together to form a ready queue. Long-term(job) scheduling is the selection of processes that will be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resources-allocation considerations, especially memory management. Short-term(CPU) scheduling is the selection of one process from the ready queue.

Operating systems must provide a mechanism for parent processes to create **new child processes**. The parent may wait for its children to terminate before proceeding, or the parent and children may execute concurrently. There are several reasons for allowing concurrent execution:

information sharing computation speedup, modularity, and convenience. The processes executing in the operating system may be either independent processes or cooperating processes. Cooperating processes require an **inter-process communication** mechanism to communicate with each other. Principally, communication is achieved through two schemes: **shared memory** and **message passing**.

A **thread** is a flow of control within a process. A **multithreaded process** contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and scalability issues such as more efficient use of multiple core.

User level threads are threads are visible to the programmer and are unknown to the **kernel.** The operating system kernel supports and manages kernel level threads. In general, user level threads are faster to create and manage than are kernel threads, as no intervention from the kernel is required.

Three different types of models relate user and **kernel threads**: the **many-to-one** model maps **any user threads** to a **single thread**. The one to one model maps each user thread to a corresponding kernel thread. The many to many model multiplexers many user threads to a smaller or equal number of kernel threads.

4.19 Review Question
Q.1 Define process and programs?
Q.2 Describe Process Control Block?
Q.3 Explain Scheduling Queues?
Q.4 Distinguish between a process and a thread?
Q.5 Differentiate various types of scheduler?
Q.6 Differentiate between user level thread and kernel level thread?
Q.7 Explain various Multithreaded Model?

Chapter 5
5.0 Memory Management
Unit Structure

- Objectives
- Introduction
- Memory Partitioning
- Swapping
- Paging
- Segmentation
- Summary
- Review Question

Objectives

- To provide a detailed description of various ways of organizing memory hardware.
- To discuss various memory-management techniques, including paging and segmentation.
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging.

Computer programs together with the data they access, must be at least partially in main memory during execution. To improve both the utilization of the CPU and the speed of its response to users, a general-purpose computer must keep several processes in memory. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the *hardware* design of the system. Most algorithms require hardware support memory is central to the operation of a modern computer system. Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore *how* a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

5.01 Memory Overview

Memory-management algorithms for multi-programmed operating systems range from the simple single-user system approach to paged segmentation. The most important determinant of the method used in a particular system is the hardware provided. Every memory address generated by the CPU must be checked for legality and possibly mapped to a physical address. The checking cannot be implemented (efficiently) in software. Hence, we are constrained by the hardware available. The various memory-management algorithms (contiguous allocation,

paging, segmentation, and combinations of paging and segmentation) differ in many aspects. In comparing different memory-management strategies, we use the following considerations:

- ✓ **Hardware support**. A simple base register or a base–limit register pair is sufficient for the single- and multiple-partition schemes, whereas paging and segmentation need mapping tables to define the address map.
- ✓ **Performance**. As the memory-management algorithm becomes more complex, the time required to map a logical address to a physical address increases. For the simple systems, we need only compare or add to the logical address—operations that are fast. Paging and segmentation can be as fast if the mapping table is implemented in fast registers. If the table is in memory, however, user memory accesses can be degraded substantially. A TLB can reduce the performance degradation to an acceptable level.
- ✓ **Fragmentation**. A multi-programmed system will generally perform more efficiently if it has a higher level of multiprogramming. For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory. To accomplish this task, we must reduce memory waste, or fragmentation. Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation. Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.
- ✓ **Relocation**. One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory in such a way that the program does not notice the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.
- ✓ **Swapping**. Swapping can be added to any algorithm. At intervals determined by the operating system, usually dictated by CPU-scheduling policies, processes are copied from main memory to a backing store and later are copied back to main memory. This scheme allows more processes to be run than can be fit into memory at one time.
- ✓ **Sharing**. Another means of increasing the multiprogramming level is to share code and data among different users. Sharing generally requires that either paging or segmentation be used to provide small packets of information (pages or segments) that can be shared. Sharing is a means of running many processes with a limited amount of memory, but shared programs and data must be designed carefully.
- ✓ **Protection**. If paging or segmentation is provided, different sections of a user program can be declared execute-only, read-only, or read–write. This restriction is necessary with shared code or data and is generally useful in any case to provide simple run-time checks for common programming errors

5.02 Basic Hardware Support

Main memory and the registers built into the processor itself are the only storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in

Memory, they must be moved there before the CPU can operate on them. Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction
that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and

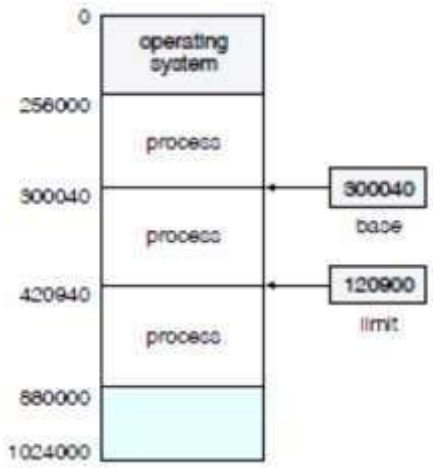
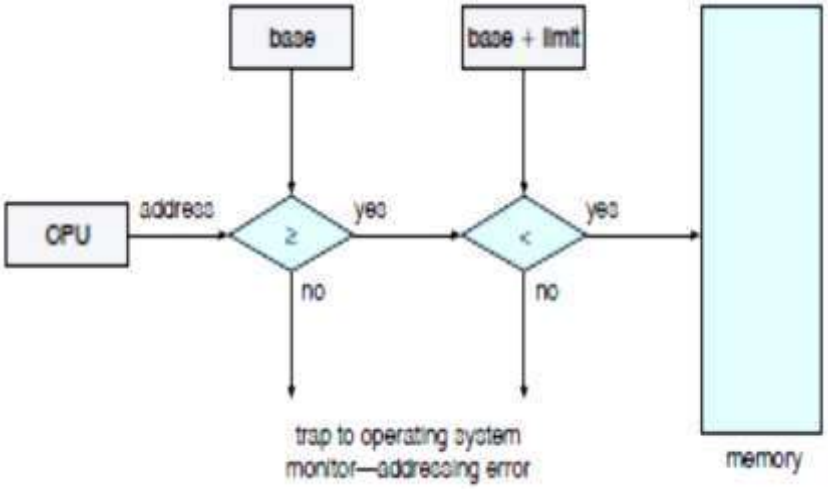


Figure 9 A base and a limit register against logical address space

main memory. A memory buffer used to accommodate a speed differential, called a **cache**. Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation to protect the operating system from access by user processes and, in addition, to protect user processes from one another. This protection must be provided by the hardware. It can be implemented in several ways.
We first need to make sure that each process has a separate memory space. To do this, we need the ability to determine the range of legal addresses that the process m ay access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 9 above. The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive) Figure 10.

Protection of memory space is accomplished by having the CPU hardware compare *every* address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other



The base and l ı uses a special privileged instr rnel mode, and since only the c ɐm can load the base and limit r This scheme al ıt prevents user programs from in kernel mode, is given unrestr This provision allows the oper

Figure 9 Hardware Address Protection

5.03 Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.

The normal procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available. Most systems allow a user process to reside in any part of the physical memory. Thus, although the address

space of the computer starts at 00000, the first address of the user process need not be 00000. This approach affects the addresses that the user program can use. In most cases, a user program will go through several steps—some of which may be optional—before being executed (Figure 11). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as *count*). A compiler will typically **bind** these symbolic addresses to re-locatable addresses (such as "14 bytes"). The linkage editor or loader will in turn bind the re-locatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses canbe done at any step along the way:

i) **Compile time**. If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.

ii) **Load time**. If it is not known at compile time where the process will reside in memory, then the compiler must generate **re-locatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

iii) **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work, as will be discussed in Section 10. Most general-purpose operating systems use this method.

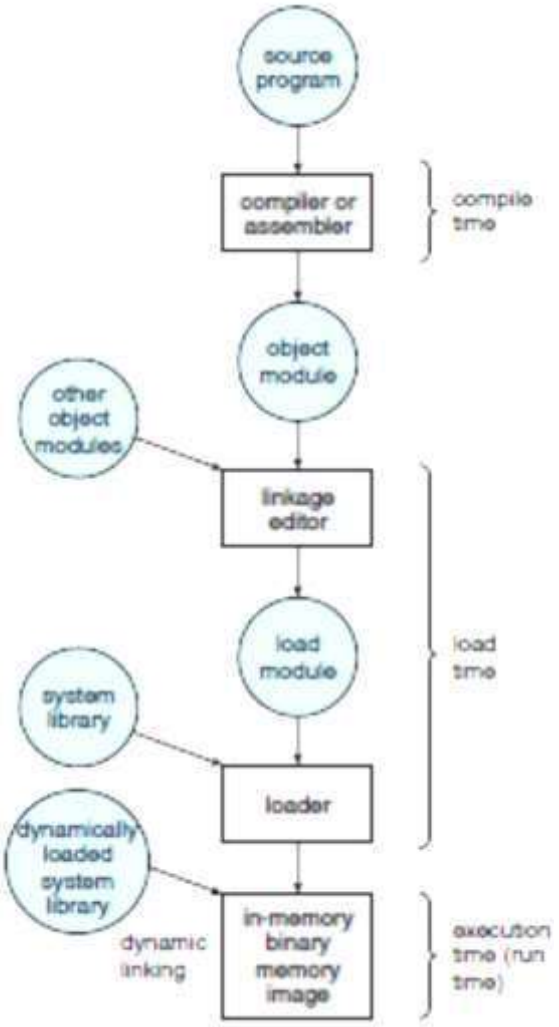


Figure 11 Multistep processing of a user program.

5.04 Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**.Weuse ***logical address*** and ***virtual address*** interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**; the set of all physical addresses corresponding to these logical addresses is a **physical address space**.

Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ. The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.We can choose from many different methods to accomplish this mapping.

5.05 Dynamic Loading

In our discussion so far, it has been necessary for the entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**. With Dynamic Linking and Shared Libraries, some operating systems support only **static linking**, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. Dynamic linking, in contrast, is similar to dynamic loading.

Here, though, linking, rather than loading, is postponed until execution time. This feature is usually used with system libraries, such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement wastes both disk space and main memory.

With dynamic linking, a *stub* is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine. Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

This feature can be extended to library updates (such as bug fixes).A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be re-linked to gain access to the new library. So that programs will not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Versions with minor changes retain the same version number, whereas versions with major changes increment the number. Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated

5.06 Swapping

A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution. For example, assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed (Figure 13). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum, it will be swapped with another process. Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready

to execute, when the CPU scheduler wants to reschedule the CPU. In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps.

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the

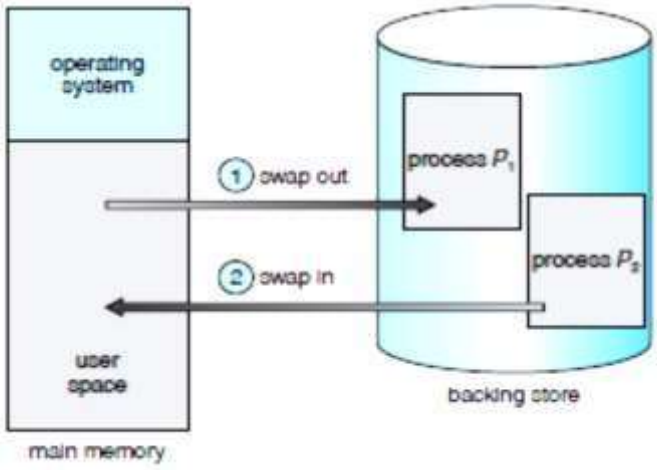


Figure 12 Swapping

5.07 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one common method, ***contiguous memory allocation***.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation in which the operating system resides in low memory. The development of the other situation is similar.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.

5.08 Memory Mapping and Protection

Before discussing memory allocation further, we must discuss the issue of memory mapping and protection. We can provide these features by using relocation register, as discussed in previous section. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address ***dynamically*** by adding the value in the relocation

register. This mapped address is sent to memory. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and other users' programs and data from being modified by this running process.

The ***relocation-register*** scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient** operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

5.09 Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree In general, as mentioned, the memory blocks available comprise a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general **dynamic storage allocation problem**, which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

• **First fit**. Allocate the *first* hole that is big enough. Searching can start either at the beginning of he set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

• **Best fit**. Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

• **Worst fit**. Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

5.10 Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous; storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, best fit for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece— the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given N allocated blocks, another $0.5\ N$ blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.Memory fragmentation can be internal as well as external

5.11 Paging

Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems.

5.12 Physical memory

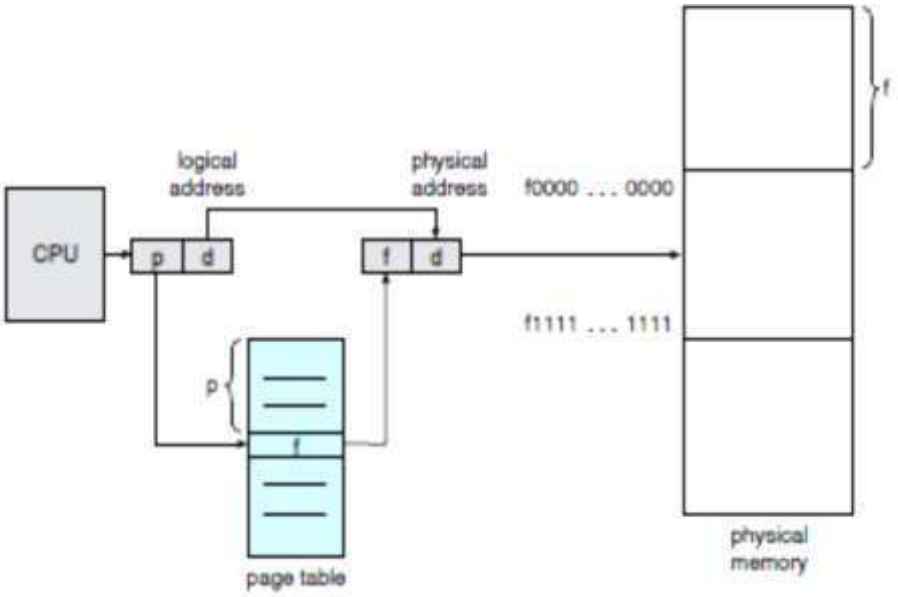


Figure 12 Paging Hardware

5.13 Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable

memory space. Some systems provide hardware, in the form of a **page-table length register (PTLR)**, to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

5.14 Shared Pages

An advantage of paging is the possibility of *sharing* common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to upport the 40 users. If the code is **reentrant code** (or **pure code**), however, it can be shared, as shown in Figure 7.13. Here we see a three-page editor—each page 50 KB in size (the large page size is used to simplify the figure)—being shared among three processes. Each process has its own data page. Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different.

Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings. Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.

5.15 Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.

If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure 13

A variation of this scheme that is favorable for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address

5.16 Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual

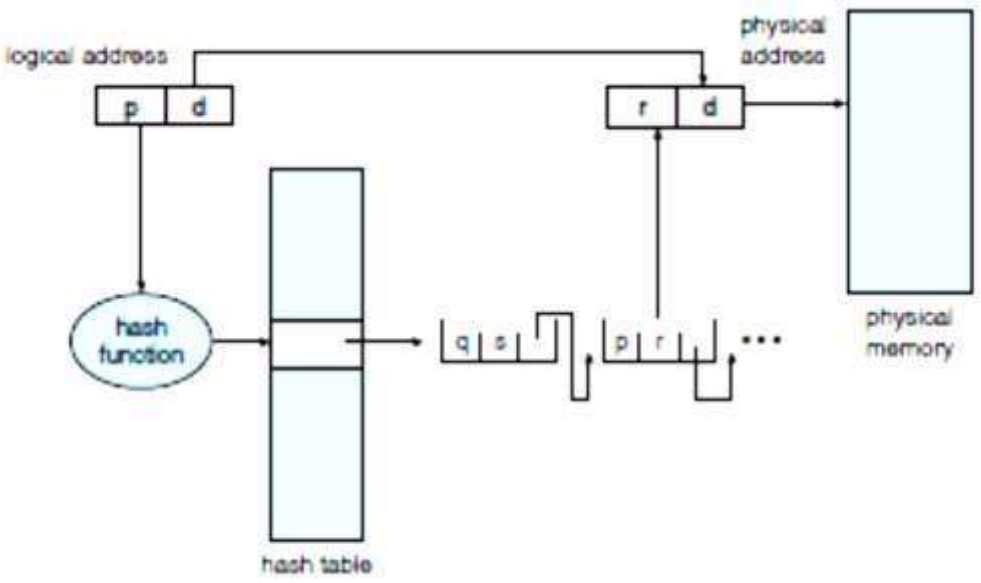


Figure 13

5.17 Segmentation

is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.)

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:* segment-number, offset>.

Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

- ✓ The code
- ✓ Global variables
- ✓ The heap, from which memory is allocated
- ✓ The stacks used by each thread
- ✓ The standard C library

5.18 Summary

Memory management algorithms for multi programmed operating systems range from the simple single user system approach to paged segmentation. The most important determinant of the method used in a particular system is the hardware provided, every memory address generated by the CPU must be checked for legality and possibly mapped to a physical address, the checking cannot be implemented in software. Hence, we are constrained by the hardware available.

The various memory management algorithms (continuous allocation, paging, segmentation, and combinations of paging and segmentation) differ in many aspects. In computing different memory management strategies, we use hardware support, performance, fragmentation, relocation, swapping, sharing and protection.

Memory-management algorithms for multi-programmed operating systems range from the simple single-user system approach to paged segmentation. The most important determinant of the method used in a particular system is the hardware provided. Every memory address generated by the CPU must be checked for legality and possibly mapped to a physical address. The checking cannot be implemented (efficiently) in software. Hence, we are constrained by the hardware available.

The various memory-management algorithms (contiguous allocation, paging, segmentation, and combinations of paging and segmentation) differ in many aspects. In comparing different memory-management strategies, we use the following considerations:

Hardware support. A simple base register or a base–limit register pair is sufficient for the single- and multiple-partition schemes, whereas paging and segmentation need mapping tables to define the address map.

Performance. As the memory-management algorithm becomes more complex, the time required to map a logical address to a physical address increases. For the simple systems, we need only compare or add to the logical address—operations that are fast. Paging and segmentation can be as fast if the mapping table is implemented in fast registers. If the table is in memory, however, user memory accesses can be degraded substantially. A TLB can reduce the performance degradation to an acceptable level.

Fragmentation. A multi-programmed system will generally perform more efficiently if it has a higher level of multiprogramming. For a given set of processes, we can increase the multiprogramming level only by packing more processes into memory. To accomplish this task, we must reduce memory waste, or fragmentation. Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation. Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.

Relocation. One solution to the external-fragmentation problem is compaction. Compaction involves shifting a program in memory in such a way that the program does not notice the change. This consideration requires that logical addresses be relocated dynamically, at execution time. If addresses are relocated only at load time, we cannot compact storage.

Swapping. Swapping can be added to any algorithm. At intervals determined by the operating system, usually dictated by CPU-scheduling policies, processes are copied from main memory to a backing store and later are copied back to main memory. This scheme allows more processes to be run than can be fit into memory at one time.

Sharing. Another means of increasing the multiprogramming level is to share code and data among different users. Sharing generally requires that either paging or segmentation be used to provide small packets of information (pages or segments) that can be shared. Sharing is a means of running many processes with a limited amount of memory, but shared programs and data must be designed carefully.

Protection. If paging or segmentation is provided, different sections of a user program can be declared execute-only, read-only, or read–write. This restriction is necessary with shared code or data and is generally useful in any case to provide simple run-time checks for common programming errors.

5.19 Revision Questions

Q. 1 Define a system that allows static linking and sharing of segments without requiring that the segment numbers be the same.

Q. 2 Describe a paging scheme that allows pages to be shared without requiring that the page numbers be the same.

Q. 3 On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to other memory? Why should it or should it not?

Q. 4 Name two differences between logical and physical addresses. Explain the difference between internal and external fragmentation. Describe a mechanism by which one segment could belong to the address space of two different processes.

Q.5 What are hardware is required for paging?

Q.6 Write a note on virtual memory?

Chapter 6

6.0 File Management

Unit Structure

- Objectives
- File Concept
- File Support
- Access Methods
- Directory Systems
- File Protection
- Free Space Management
- Summary
- Review Questions

Objective

- To explain the function of file systems.
- To describe the interfaces to the file systems.
- To discuss file system design tradeoffs, including access methods, file sharing; file locking, and directory structure.
- To explore file system protection.

6.01 File Concept

A file is a collection of similar records. The file is treated as a single entity by users and applications and may be referred by name. Files have unique file names and may be created and deleted. Restrictions on access control usually apply at the file level.A file is a container for a collection of information. The file manager provides a protection mechanism to allow users administrator how processes executing on behalf of different users can access the information in a file. File protection is a fundamental property of files because it allows different people to store their information on a shared computer.

File represents programs and data. Data files may be numeric, alphabetic, binary or alpha numeric. Files may be free form, such as text files. In general, file is sequence of bits, bytes, lines or records.

A file has a certain defined structure according to its type. This includes:

- ✓ Text File
- ✓ Source File
- ✓ Executable File
- ✓ Object File

6.02 File Structure

Four terms are use for files. The are:

- ✓ Field
- ✓ Record
- ✓ Database

A **field** is the basic element of **data**. An individual field contains a single value. A record is a collection of related fields that can be treated as a unit by some application program.

A ***file*** is a collection of similar records. The file is treated as a singly entity by users and applications and may be referenced by name. Files have file names and maybe created and deleted. Access control restrictions usually apply at the file level.

A **database** is a collection of related data. Database is designed for use by a number of different applications. A database may contain all of the information related to an organization or project, such as a business or a scientific study. The database itself consists of one or more types of files. Usually, there is a separate database management system that is independent of the operating system.

6.03 File Attributes

File attributes vary from one operating system to another. The common attributes are:

- **Name** – only information kept in human-readable form.
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring

  Information about files are kept in the directory structure, which is maintained on the disk

6.04 File Operations

Any file system provides not only a means to store data organized as files, but a collection of functions that can be performed on files. Typical operations include the following:

- **Create:** A new file is defined and positioned within the structure of files.

- **Delete:** A file is removed from the file structure and destroyed.
- **Open:** An existing file is declared to be "opened" by a process, allowing the process to perform functions on the file.
- **Close:** The file is closed with respect to a process, so that the process no longer may perform functions on the file, until the process opens the file again.
- **Read:** A process reads all or a portion of the data in a file.
- **Write:** A process updates a file, either by adding new data that expands the size of the file or by changing the values of existing data items in the file.

6.05 File Types – Name, Extension

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts: a ***name*** and ***an extension***. Following table gives the file type with usual extension and function.

File Type	Usual Extension	Function
Executable	exe, com, bin	Read to run machine language program.
Object	obj, o	Compiled, machine language, not linked
Source Code	c, cc, java, pas asm, a	Source code in various language
Text	txt, doc	Textual data, documents

6.06 File Management Systems:

A file management system is that set of system software that provides services to users and applications in the use of files. Following objectives for a file management system:

- ✓ To meet the data management needs and requirements of the user which include storage of data and the ability to perform the aforementioned operations.
- ✓ To guarantee, to the extent possible, that the data in the file are valid.
- ✓ To optimize performance, both from the system point of view in terms of overall throughput.
- ✓ To provide I/O support for a variety of storage device types.
- ✓ To minimize or eliminate the potential for lost or destroyed data.
- ✓ To provide a standardized set of I/O interface routines to use processes.
- ✓ TO provide I/O support for multiple users, in the case of multiple-user systems

6.07 File System Architecture.

At the lowest level, **device drivers** communicate directly with peripheral devices or their controllers or channels. A device driver is responsible for starting I/O operations on a device and processing the completion of an I/O request. For file operations, the typical devices controlled are disk and tape drives. Device drivers are usually considered to be part of the operating system. The ***I/O control***, consists of device drivers and interrupt handlers to transfer information between the memory and the disk system. A device driver can be thought of as a translator.

The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. The file-organization module knows about files and

their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to ***physical block*** addresses for the basic file system to transfer.

Each file's logical blocks are numbered from 0 (or 1) through N, whereas the physical blocks containing the data usually do not match the logical numbers, so a translation is needed to locate each block. The file-organization module also includes the free-space manager, which tracks unallocated and provides these blocks to the file organization module when requested.

The logical file system uses the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. The logical file system is also responsible for protection and security.
To create a new file, an application program calls the ***logical file system***. The logical file system knows the format of the directory structures. To create a new file, it reads the appropriate directory into memory, updates it with the new entry, and writes it back to the disk. Once the file is found the associated information such as ***size***, ***owner***, ***access permissions*** and ***data block*** locations are generally copied into a table in memory, referred to as the open-file fable, consisting of information about all the currently opened files.

The first reference to a file (normally an open) causes the directory structure to be searched and the directory entry for this file to be copied into the table of opened files. The index into this table is returned to the user program, and all further references are made through the index rather than with a symbolic name. The name given to the index varies. **Unix** systems refer to it as a file descriptor, Windows/NT as a file handle, and other systems as a file control block.

Consequently, as long as the file is not closed, all file operations are done on the open-file table. When the file is closed by all users that have opened it, the updated file information is copied back to the disk-based directory structure.

6.08 File-System Mounting
As a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. The mount procedure is straight forward. The stem is given the name of the device, and the location within the file structure at which to attach the file system (called ***the mount point***).

The operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate.

6.09 Allocation Methods

The direct-access nature of disks allows us flexibility in the implementation of files. Three major methods of allocating disk space are in wide use: contiguous, linked and indexed. Each method has its advantages and disadvantages.

6.10 Contiguous Allocation

The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. Notice that with this ordering assuming that only one job is accessing the disk, accessing block **b + 1** after block b normally requires no head movement.

When head movement is needed, it is only one track. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal.

Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long, and starts at location!), then it occupies blocks **b, b + 1, b + 2, ..., b + n – 1**.

The directory entry for each file indicates the address of the starting block and the length of the area allocate for this file.

Accessing a file that has been allocated ***contiguously*** is easy. For ***sequential access***, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block **b + i.** The contiguous disk-space-allocation problem can be seen to be a particular application of the general ***dynamic storage-allocation*** .**First Fit and Best Fit** are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first-fit and best-fit are more efficient than worst-fit in terms of both time and storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.

These ***algorithms*** suffer from the problem of ***external fragmentation***. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunks is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be either a minor or a major problem.

To prevent loss of significant amounts of disk space to external fragmentation, the user had to run repacking routine that copied the entire file system onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk by allocating contiguous space from

this one large hole. This scheme effectively compacts all free space into one contiguous space, solving the ***fragmentation*** problem. The cost of this compaction is time.

The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. During this down time, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines. A major problem is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. The user will normally over estimate the amount of space needed, resulting in considerable wasted space.

6.11 Linked Allocation

Linked allocation solves all problems of contiguous allocation. With link allocation, each file is a linked list disk blocks; the disk blocks may be scattered anywhere on the disk. This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free bio to be found via the free-space management system, and this new block is the written to, and is linked to the end of the file

There is no external fragmentation with linked allocation, and any free! block on the free-space list can be used to satisfy a request. Notice also that there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks. Consequently, it is never necessary to compact disk space.

The major problem is that it can be used effectively for only sequential access files. To find the ith block of a file we must start at the beginning of that file, and follow the pointers until we get to the ith block. Each access to a pointer requires a disk read and sometimes a ***disk seek.*** Consequently, it is inefficient to support a ***direct-access*** capability for linked allocation files.

Linked allocation is the space required for the pointers If a pointer requires 4 bytes out of a 512 Byte block then 0.78 percent of the disk is being used for pointer, rather than for information.

The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate the clusters rather than ***blocks***. For instance, the file system defines a cluster as 4 blocks and operates on the disk in only cluster units.

Pointers then use a much smaller percentage of the file's disk space. This method allows the ***logical-to-physical block mapping*** to remain simple, but improves disk throughput (fewer disk head seeks) and decreases the space needed for block allocation and ***free-list*** management. The cost of this approach is an increase in internal fragmentation.

Yet another problem is ***reliability***. Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer— were lost or damaged. Partial solutions are to use doubly linked lists or to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

An important variation, on the linked allocation method is the use of a **file allocation table (FAT).** This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each-partition is set aside to contain the table. The table has one entry for each disk block, and is indexed by block number. The **FAT** is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value -as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-offile value. An illustrative example is the **FAT** structure of for a file consisting of disk blocks 217, 618, and 339.

6.12 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. The absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order Indexed allocation solves this problem by bringing all the pointers together into one location: the ***index block***.

Each file has its own index block, which is an array of disk-block addresses. The **ith** entry in the index block points to the **ith** block of the file. The directory contains the address of the index block. When the file is created, all pointers in the index block are set to nil. When the ***ith*** block is first written, a block is obtained: from the free space manager and its address- is put in the ***ith*** index-block entry.

Allocation supports ***direct access***, without suffering from external fragmentation because any free block on he disk may satisfy a request for more space. Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

1. **Linked scheme.** An index block is normally one disk block. Thus, it can be read and written directly by itself.

2. **Multilevel index.** A variant of the linked representation is to use a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block, and that block to find the desired data block.

6.13 Free-Space Management

Since there is only a limited amount of disk space, it is necessary to reuse the space from deleted files for new files, if possible.

Bit Vector

Free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0. For example consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be 00111100111111000110000001110000 0 …..

The main advantage of this approach is that it is relatively simple and efficient to find the first free block or n consecutive free blocks on the disk. The calculation of the block number is (number of bits per word) x (number of 0-value words) + offset of first 1 bit

Linked List

Another approach is to link together all the free disk blocks, keeping a pointy to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used.

Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 of these blocks are actually free. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

Counting

Several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous allocation algorithm or through clustering. A list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block.

Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as count is generally greater than 1.

6.14 Summary

A file is an abstract data type defined and implemented by the operating system. It is a sequence of logical records. A logical record may be a byte, a line, or a more complex data item. The operating system may specifically support various record types or may leave that support to the application program.

The major task for the operating system is to map the logical file concept onto physical storage devices such as magnetic tape or disk. All the file information kept in the directory structure. File system is implemented on the disk and memory.

Disk address defines a linear addressing on the disk. **Continuous allocation algorithm** suffers from the external fragmentation. Free space management techniques are bit vector, linked list. Grouping, counting.

Since files are the main information storage mechanism in most computer systems, file protection is needed. Access to files can be controlled separately for each type of access – read, write, execute, append, delete, list directory, and so on. File protection can be provided by access lists, passwords, or other techniques.

6.15 Revision Questions

Q.1 Explain the file concept?

Q.2 List the different types of files?

Q.3 Describe the different types of access methods?

Q.4 What are the advantages and disadvantages of continuous, linked and indexed allocation methods?

Q.5 Explain the file system structure?

Q.6 What are different types of partitions and mounting?

Chapter 7

7.0 Input Output Hardware

Unit Structure

- Objectives
- Principal of I/O Hardware
- Polling
- I/O Devices
- Device Controllers
- Direct Memory Access
- Revision Questions

Objectives

- Explore the structure of an operating system‘s I/O subsystem.
- Discuss the principles of I/O hardware and its complexity.

I/O Hardware

Computers operate a great many kinds of devices. General types include storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse).

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point termed a port (for example, a serial port). If one or more devices use a common set of wires, the connection is called a bus.

When device A has a cable that plugs into device B, and device B has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a daisy chain. It usually operates as a bus.

A controller is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is an example of a simple device controller. It is a single chip in the computer that controls the signals on the wires of a serial port.

The **SCSI** bus controller is often implemented as a separate circuit board (a host adapter) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers.

An **I/O port** typically consists of four registers, called the status, control, data-in, and data-out registers. The status register contains bits that can be read by the host. These bits indicate states such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether there has been a device error. The control register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in

the control register of a serial port chooses between full-duplex and half-duplex communication, another enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.

The data-in register is read by the host to get input, and the data out register is written by the host to send output. The data registers are typically 1 to 4 bytes. Some controllers have **FIFO** chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

7.01 POLLING

Incomplete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple. The controller indicates its state through the busy bit in the status register. (Recall that to set a bit means to write a 1 into the bit, and to clear a bit means to write a 0 into it.)

The controller sets the busy bit when it is busy working, and clears the busy bit when it is ready to accept the next command. The host signals its wishes via the command-ready bit in the command register. The host sets the command-ready bit when a command is available for the controller to execute.

For this example, the host writes output through a port, coordinating with the controller by handshaking as follows:

- The host repeatedly reads the busy bit until that bit becomes clear.
- The host sets the write bit in the command register and writes a byte into the data-out register.
- The host sets the command-ready bit.
- When the controller notices that the command-ready bit is set, it sets the Busy.
- The controller reads the command register and sees the write command.
- It reads the data-out register to get the byte, and does the I/O to the device.
- The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.
- The host is busy-waiting or polling: It is in a loop, reading the status register over and over until the busy bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task

7.02 I/O Devices

Categories of I/O Devices :

- ✓ Human readable
- ✓ Machine readable
- ✓ Communication

1. **Human Readable** is suitable for communicating with the computer user. Examples are printers, video display terminals, keyboard etc.
2. **Machine Readable** is suitable for communicating with electronic equipment. Examples are disk and tape drives, sensors, controllers and actuators.
3. **Communication** is suitable for communicating with remote devices. Examples are digital line drivers and modems.

Differences between I/O Devices

1. **Data rate** : there may be differences of several orders of magnitude between the data transfer rates.
2. **Application:** Different devices have different use in the system.
3. **Complexity of Control**: A disk is much more complex whereas printer requires simple control interface.
4. **Unit of transfer**: Data may be transferred as a stream of bytes or characters or in larger blocks.
5. **Data representation**: Different data encoding schemes are used for different devices.
6. **Error Conditions**: The nature of errors differs widely from one device to another.

7.03 Direct Memory Access

A special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called **Direct Memory Access** (DMA).

DMA can be used with either polling or interrupt software. DMA is particularly useful on devices like disks, where many bytes of information can be transferred in single I/O operations. When used in conjunction with an interrupt, the CPU is notified only after the entire block of data has been transferred. For each byte or word transferred, it must provide the memory address and all the bus signals that control the data transfer. Interaction with a device controller is managed through a device driver.

Device drivers are part of the operating system, but not necessarily part of the **OS kernel**. The operating system provides a simplified view of the device to user applications (e.g., character devices vs. block devices in UNIX). In some operating systems (e.g., Linux), devices are also accessible through the /dev file system. In some cases, the operating system buffers data that are transferred between a device and a user space program (disk cache, network buffer). This usually increases performance, but not always.

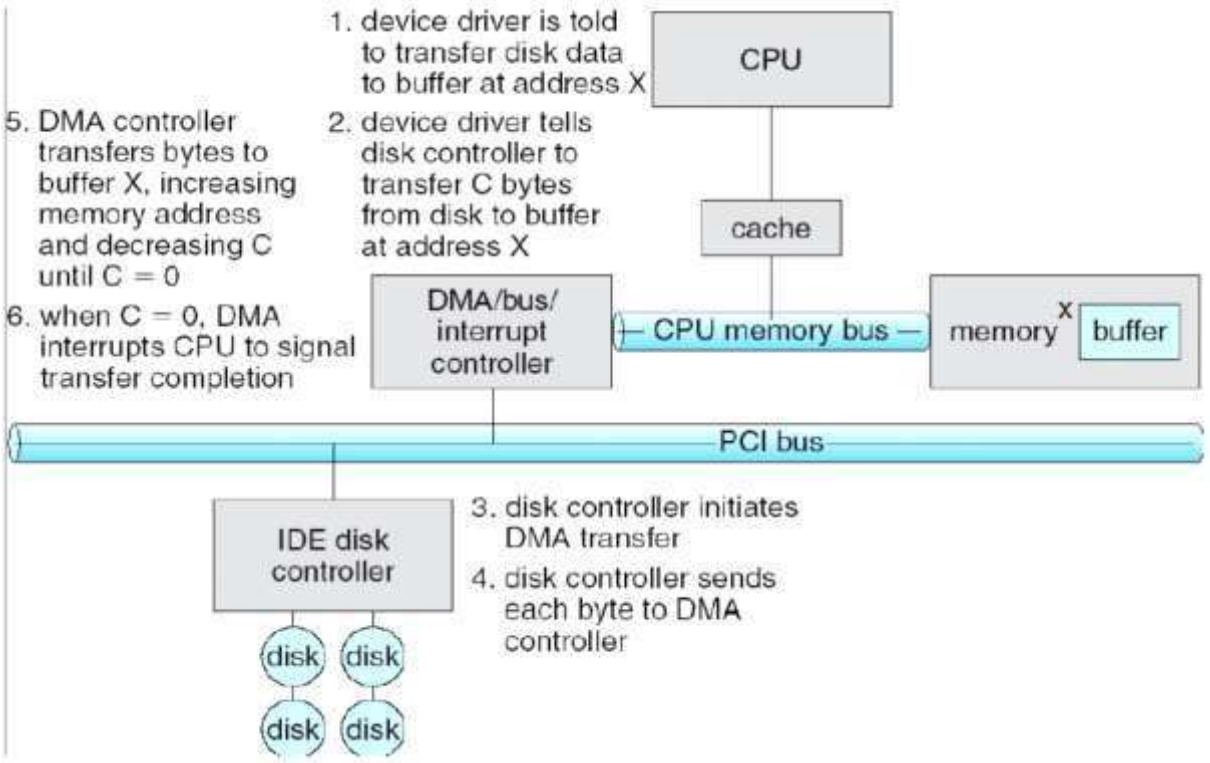
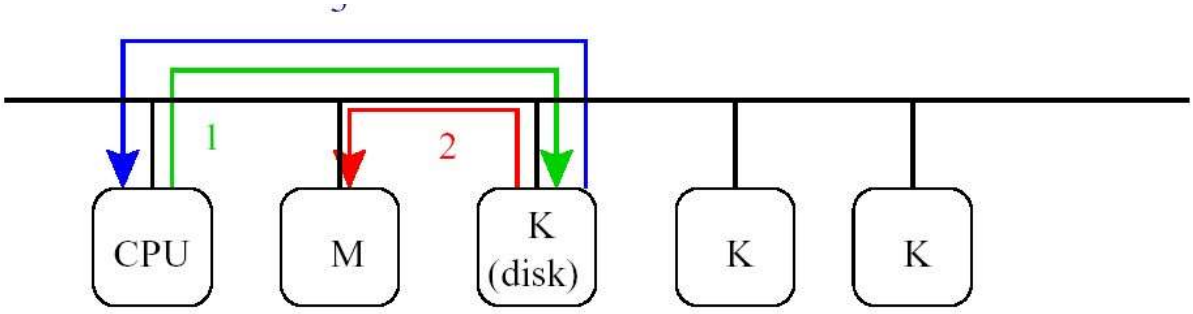


Figure 15 DMA



1. CPU issues DMA request to controller
2. controller directs data transfer
3. controller interrupts CPU

Figure 16

7.04 Device Controllers

A computer system contains a multitude of I/O devices and their respective controllers:

- network card
- graphics adapter
- disk controller
- DVD-ROM controller
- serial port
- USB
- sound card

7.05 Summary

The basic hardware elements involved in I/O buses, device controllers, and the device themselves. The work of moving data between devices and main memory is performed by the CPU as programmed I/O or is offloaded to a DMA controller. The kernel module that controls a device driver.

The **system call** interface provided to applications is designed to handle several basic categories of hardware, including block devices, character devices; memory mapped files, network sockets, and programmed interval timers. The system calls usually block the processes that issue them, but non blocking and asynchronous calls are used by the kernel itself and by applications that must not sleep while waiting for an I/O operation to complete.

The kernel‘s I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, caching. **Spooling**, device reservation, and error handling.

7.06 Review Questions

Q.1 Define and Explain Direct Memory Access?

Q.2 Explain device controllers?

Q.3 Explain concept of I/O Hardware?

Q.4 Explain concept of Polling?

Q.5 Explain I/O devices?

CHAPTER 8

8.0 I/O SOFTWARE

Unit Structure

- Objectives
- Principle of I/O Software
- Interrupts
- Application I/O Interfaced
- Clocks and Timers
- Blocking and Non-blocking I/O
- Kernel I/O Subsystem
- Scheduling
- Buffering
- Caching
- Spooling and Device Reservation
- Error Handling
- Device Drivers
- Summary
- Model Question

Objective

- ❖ Explore the structure of an operating system‘s I/O subsystem.
- ❖ Discuss the principles of I/O software.
- ❖ Provide details of the performance aspects of I/O software.

8.01 PRINCIPLES OF I/O SOFTWARE

8.0.1Interrupts

The CPU hardware has a wire called the interrupt request line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction pointer, and jumps to the interrupt-handler routine at a fixed address in memory.

The **interrupt handler** determines the cause of the interrupt, performs the necessary processing, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches to the interrupt handler, and the handler clears the interrupt by servicing the device. Figure 17 summarizes the interrupt-driven I/O cycle.

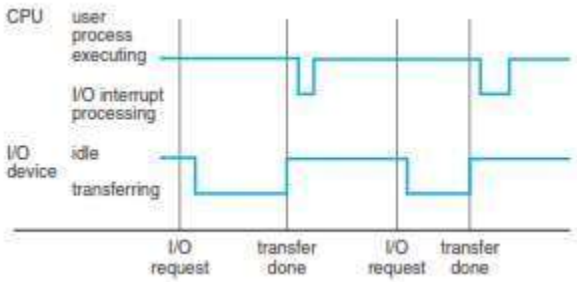


Figure 17 Interrupt Driven I/O

This basic interrupt mechanism enables the CPU to respond to an asynchronous event, such as a device controller becoming ready for service. In a modern operating system, we need more sophisticated interrupt-handling features.

First, we need the ability to defer interrupt handling during critical processing. Second, we need an efficient way to dispatch to the proper interrupt handler for a device, without first polling all the devices to see which one raised the interrupt. Third, we need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts, and can respond with the appropriate degree of urgency. In modern computer hardware, these three features are provided by the CPU and by the interrupt-controller hardware.

CPUs have two interrupt request lines. One is the **non-maskable** interrupt, which is reserved for events such as unrecoverable memory errors. The second interrupt line is **maskable.** It can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

This address is an offset in a table called the interrupt vector. This vector contains the memory addresses of .specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.

The interrupt mechanism also implements a system of interrupt priority levels. This mechanism enables the CPU to defer the handling of low-priority interrupts without masking off all interrupts, and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt.
The interrupt mechanism is also used to handle a wide variety of exceptions, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode.

A **system call** is a function that is called by an application to invoke a kernel service. The system call checks the arguments given by the application, builds a data structure to convey the arguments to the kernel, and then executes a special instruction called a software interrupt, or a trap.

Interrupts can also be used to manage the flow of control within the kernel. If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes. Consequently, the kernel code that completes a disk read is implemented by a pair of interrupt handlers. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low-priority interrupt to complete the work. The corresponding handler completes the user level I/O by copying data from kernel buffers to the application space and then by calling the scheduler to place the application on the ready queue.

8.02 Application I/O Interfaced

Structuring techniques and interfaces for the operating system enable I/O devices to be treated in a standard, uniform way. For instance, how an application can open a file on a disk without knowing what kind of disk it is, and how new disks and other devices can be added to a computer without the operating system being disrupted.

The actual differences are encapsulated in ken modules called device drivers mat internally are custom tailored to each device but that export one of the standard interfaces.

The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls.

- **Character-stream or block**. A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
- **Sequential or random-access**. A sequential device transfers data in a fixed order that is determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.
- **Synchronous or asynchronous.** A synchronous device is one that performs data transfers with predictable response times. An asynchronous device exhibits irregular or unpredictable response times.
- **Sharable or dedicated.** A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.
- **Speed of operation.** Device speeds range from a few bytes per second to a few gigabytes per second.
- **Read-write, read only, or write only.** Some devices perform both input and output, but others support only one data direction. For the purpose of application access, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types.

Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer. The performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the **read-write-seek** interface used for disks. .

8.03 Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

1. Give the current time

2. Give the elapsed time

3. Set a timer to trigger operation X at time T

These functions are used heavily by the operating system, and also by time sensitive applications. The hardware to measure elapsed time and to trigger operations is called a programmable interval timer.

8.04 Blocking and Non-blocking I/O

One remaining aspect of the system-call interface relates to the choice between blocking I/O and non-blocking (asynchronous) I/O. When an application calls a blocking system call, the execution of the application is suspended. The application is moved from the operating system's run queue to a wait queue.

After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call.

Some user-level processes need non-blocking I/O.

8.05 Kernel I/O Subsystem

Kernels provide many services related to I/O. The services that we describe are I/O scheduling, buffering caching, spooling, device reservation, and error handling.

Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. Operating-system developers implement scheduling by maintaining a queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device.

The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications.

Buffering

A buffer is a memory area that stores data while they are transferred between two devices or between a device arid an application. Buffering is done for three reasons.

One reason is to cope with a speed mismatch between the producer and consumer of a data stream.

Second buffer while the first buffer is written to disk. A second use of buffering is to adapt between devices that have different data transfer sizes.

A third use of buffering is to support copy semantics for application I/O.

8.06 Caching

A cache is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.

Caching and buffering are two distinct functions, but sometimes a region of memory can be used for both purposes.

8.07 Spooling and Device Reservation

A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time.

In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in kernel thread.

8.08 Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors.

Device drivers

In computing, a device driver or software driver is a computer program allowing higher-level computer programs to interact with a hardware device.

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device.

Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific.

They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

8.09 Summary

The system call interface provided to applications is designed to handle several basic categories of hardware, including block devices, character devices, memory mapped files, network sockets, and programmed interval timers. The system calls usually block the processes that issue them, but non blocking and asynchronous calls are used by the kernel itself and by applications that must not sleep while waiting for an I/O operation to complete.

The kernel‘s I/O subsystem provides numerous services. Among these are I/O scheduling, buffering, caching. Spooling, device reservation, and error handling. I/O system calls are costly in terms of CPU consumption because of the many layers of software between a physical device and an application.

8.10 Review Questions

Q.1 Explain application of I/O interface?

Q.2 Describe blocking and non blocking I/O?

Q.3 Explain following concepts

a) Clock and Timers

b) Device Drivers

Q.4 Write a short notes on

a) Scheduling b) Buffering

c) Error Handling

Q.5 Explain spooling mechanism?

Q.6 Explain Caching in details?

Appendix1: SAMPLE EXAMINATION PAPER 1

Mount Kenya University

UNIVERSITY EXAMINATION 2012/2013

SCHOOL OF PURE AND APPLIED SCIENCES DEPARTMENT

OF INFORMATION TECHNOLOGY BACHELOR OF

SCIENCE IN INFORMATION TECHNOLOGY VIRTUAL

CAMPUS

UNIT CODE: BIT2104 UNIT TITLE: OPERATING SYSTEMS 1

DATE: AUGUST 20XX MAIN TIME: 2 HOURS

Instructions: Answer Question ONE and any other TWO.

Question One

(a) Define the term operating system. (2 Marks)

(b) List FOUR functions of an operating system. (4 Marks)

(c) What is the distinguishing characteristic of real time operating system? (2 Marks)

(d) Discuss the role of the following schedulers in process control:

(i) Short term scheduler (3 Marks)
(ii) Medium term scheduler (3 Marks)
(iii) Long term scheduler (3 Marks)

(e) Below is a set of process available for execution in a multi-programmed environment:

Process	Burst Time	Arrival Time
A	12	0
B	6	1
C	2	2
D	4	3
E	9	6

Schedule the processes using First Come First served (FCFS) and shortest Remaining Time Next (SRTN) and comment on your average turn around time and the average waiting time.
(6 Marks)

(f) Show how each of the FOUR policy conditions for deadlocks can be avoided.
(4 Marks)

(g) What is inferred when a section of code is described as a critical section?
(3 Marks)

Question Two

(a) Define the term process. (2 Marks)

(b) Describe the possible states of a process and the transitions between them.
(7 Marks)

(c) Describe briefly main memory management based on the following:

(i) Base and limit registers (3 Marks)
(ii) Memory fixed partition (3 Marks)
(iii) Swapping (3 Marks)

(d) Explain the term deadlock in process management. (2 Marks)

Question Three

(a) List and explain the conditions sufficient and necessary to produce deadlock?
(4 Marks)

(b) Identify three strategies that can be used to deal with deadlocks. (6 Marks)

(c) Describe the following disk scheduling algorithms:

(i) First come First served (FCFS)
(ii) Shortest Seek Time First (SSTF)
(iii) SCAN (10 Marks)

Question Four

(a) Describe briefly the following concepts based on operating system structure.
(i) Layered structure
(ii) Virtual machine
(iii) Micro-Kernel structure
(9 Marks)

(b) Discuss the following file allocation methods citing one advantage and one disadvantage of each method.
(i) Contiguous allocation (4 Marks)
(ii) Linked allocation (4 Marks)
(c)Explain the advantages of a multi-level queue in CPU scheduling. (3 Marks)

Appendix2: SAMPLE EXAMINATION PAPER 2

Mount Kenya University

UNIVERSITY EXAMINATION 2012/2013

SCHOOL OF PURE AND APPLIED SCIENCES DEPARTMENT

OF INFORMATION TECHNOLOGY BACHELOR OF

SCIENCE IN INFORMATION TECHNOLOGY VIRTUAL

CAMPUS

UNIT CODE: BIT2104 UNIT TITLE: OPERATING SYSTEMS 1

DATE: AUGUST 20XX MAIN TIME: 2 HOURS

Instructions: Answer Question ONE and any other TWO.

Question One

a) Explain briefly three main functions of an operating system in a computer. (6mks)

b) State and explain briefly two utility programs found in an operating system. (4mks)

c) Describe briefly main memory management based on the following. (6mks)
   (i) Paging
   (ii) Relocation and swapping

d) Describe the difference between external and internal fragmentation. (4mks)

e) Describe any three goals a scheduling policy must meet. (6mks)

f) Explain an advantage of multiprogramming. (2mks)

g) Describe the concept of inter-process communication and synchronization. (2mks)

Question Two

a) Explain what is meant by the term critical-section as used in operating system. (2mks)

b) Define the term deadlock. (2mks)

c) Discuss four conditions that are necessary for deadlocks to occur in an operating systems. (8mks)

d) Briefly describe the function of a device drivers. (2mks)

e) Giving examples, briefly discuss the following scheduling mechanism. (4mks)

   (i) Preemptive
   (ii) Non preemptive

Question Three

a) State and briefly explain the two types of 1/0 devices. (4mks)

b) State the 1/0 software design issues. (2mks)

c) Briefly explain the following scheduling policies.
   (i) First come serve algorithm
   (ii) Shortest job first algorithm
   (iii) Round robin algorithm

d) State four circumstances under which CPU scheduling decision may take place. (4mks)

e) Explain the following terms as used in scheduling criterion. (4mks)
   (i) CPU utilization
   (ii) Turnaround time
   (iii) Waiting time
   (iv) Response time

Question Four

a) Explain three memory management techniques. (6mks)

b) State and briefly explain two file structures. (4mks)

c) Outline six operations performed on a files. (6mks)

d) Define a process? (2mks)

e) Distinguish between deadlock avoidance al deadlock prevention. (2mks)