APPLIED OPERATING SYSTEM (AOS)

for

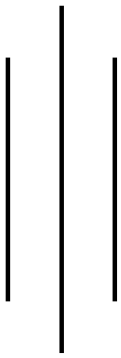
Bachelor of Engineering in Information Technology (BEIT)

and

Bachelor of Engineering in Software Engineering (BESE)

V SEMESTER

POKHARA UNIVERSITY



Prepared by

Asst. Prof. Madan Kadariya

HOD, Department of Information Technology Engineering

Nepal College of Information Technology

Balkumari Lalitpur, Nepal

Table of Contents

TABLE OF CONTENTS ..... **I**

LIST OF FIGURES ..... **VII**

LIST OF TABLES ..... **IX**

OPERATING SYSTEM TYPES AND STRUCTURE **1**

1.1 INTRODUCTION 1
1.2 OPERATING SYSTEM 1
1.2.1 Definition of Operating System: *1*
1.2.2 Functions of Operating System *1*
1.2.3 Operating System as User Interface *1*
1.3 I/O SYSTEM MANAGEMENT 3
1.4 ASSEMBLER 3
1.5 COMPILER 3
1.6 LOADER 3
1.7. HISTORY OF OPERATING SYSTEM 3
1.8 INTRODUCTION TO OPERATING SYSTEM 4
1.8.1. Operating System Services *4*
1.8.2. Operating System Components *5*
1.8.3. Computer Startup *5*
1.8.4. Computer System Organization *6*
1.8.5. Computer-System Operation *6*
1.8.6. Common Functions of Interrupts *6*
1.8.7. I/O Structure *6*
1.8.8. Direct Memory Access Structure *7*
1.8.9. Storage Structure *7*
1.8.10. Storage Hierarchy *7*
1.9. COMPUTER-SYSTEM ARCHITECTURE 7
1.10 SYMMETRIC MULTIPROCESSING ARCHITECTURE 8
1.11. A DUAL-CORE DESIGN 8
1.12. CLUSTERED SYSTEMS 8
1.13. OPERATING-SYSTEM OPERATIONS 9
1.13.1. Transition from User to Kernel Mode *9*
1.14. I/O SUBSYSTEM 9
1.15. PROTECTION AND SECURITY 10
1.16. COMPUTING ENVIRONMENTS 10
1.17. PEER-TO-PEER COMPUTING 10
1.18. WEB-BASED COMPUTING 11
1.19. OPEN-SOURCE OPERATING SYSTEMS 11
1.20. BATCH SYSTEM 11
1.21. TIME SHARING SYSTEMS 12
1.22. MULTIPROGRAMMING 13
1.23. SPOOLING 14
1.24. PARALLEL SYSTEMS 14
1.25. REAL TIME SYSTEMS 15
1.26. DISTRIBUTED SYSTEMS 15
1.27. ESSENTIAL PROPERTIES OF THE OPERATING SYSTEM 16

1.28. SYSTEM COMPONENTS 16
1.28.1. Process Management *16*
1.28.2. Main Memory Management *17*
1.28.3. File Management *17*
1.28.4. Input-Output Management *17*
1.28.5. Secondary Storage Management *17*
1.28.6. Caching *17*
1.28.7. Networking *17*
1.28.8. Protection *18*
1.28.9. Command Interpreter System *18*
1.29. OS SERVICES 18
1.29.1. Services Provided to Users *18*
1.29.2. Services provided to system *18*
1.30. SYSTEM CALLS 19
1.30.1. System Call Implementation *20*
1.30.2. System Call Parameter Passing *21*
1.31. SYSTEM PROGRAMS 22
1.32. SYSTEM STRUCTURE 22
1.32.1. Simple Structure *22*
1.32.3. Layered Structure *23*
1.33. MICROKERNEL STRUCTURE 24
1.34. CLIENT SERVER STRUCTURE 25
1.35. VIRTUAL MACHINES 25
1.36. MODULES STRUCTURE 26
1.36.1. Mechanisms and Policies *26*
1.37. SYSTEM IMPLEMENTATION 26
1.38. SYSTEM GENERATION (SYSGEN) 27

SUMMARY 27

PROCESS MANAGEMENT/THREAD MANAGEMENT 30
2.1. CONCEPT OF PROCESS 30
2.1.1 Processes and Programs *30*
2.2 PROCESS STATE 30
2.2.1 Suspended Processes *31*
2.3 PROCESS CONTROL BLOCK (PCB) 31
2.4 PROCESS MANAGEMENT / PROCESS SCHEDULING 33
2.4.1. Scheduling *33*
2.4.2 Scheduling Queues *33*
2.4.3 Schedules *35*
2.5 CONTEXT SWITCH 36
2.6 OPERATION ON PROCESSES 37
2.6.1. Create Process *37*
2.6.2 Terminate a Process *38*
2.7 CO-OPERATING PROCESSES 39
2.7.1. Interprocess Communication (IPC) *39*
2.8 THREAD MANAGEMENT 43
2.8.1. Introduction of Thread *43*
2.8.2. Multithreaded Server Architecture *43*

2.8.3. WHY THREADS? (BENEFITS) 44
2.8.4. Process vs Thread *44*
2.8.5 Types of Thread *44*
2.8.6 Multithreading Models *46*
2.8.7 Difference between User Level & Kernel Level Thread *47*
2.8.8. Threading Issues *48*
2.9. PROCESS SCHEDULING 50
2.9.1. CPU Scheduler *50*
2.9.2. Preemptive and non-preemptive Scheduling *50*
2.9.3. Dispatcher *51*
2.9.4. Scheduling Criteria *51*
2.9.5. Optimization Criteria *52*
2.9.6. Scheduling Algorithms Goals *52*
2.9.7. Job (Process) Scheduling Algorithms *53*
2.9.8. Time Quantum and Context Switch Time *60*
2.9.9. Turnaround Time Varies with the Time Quantum *60*
2.9.10 Multilevel Queue Scheduling *61*
2.9.11. Multilevel Feedback Queue *61*
2.9.12. Example of Multilevel Feedback Queue *62*
2.9.13. Thread Scheduling *62*
2.10. PROCESS SYNCHRONIZATION 64
2.10.1 Principle of Concurrency *64*
2.10.2 Producer-Consumer Problem *64*
2.10.3 Race Condition *66*
2.10.4 Avoiding Race Conditions: *67*
General structure of process *69*
Peterson's Solution (Two Task Solutions (Initial Attempts) *69*
Algorithm 1 ..... *70*
Algorithm 2 ..... *70*
Algorithm 3 (Peterson's Algorithm) *70*
2.10.5. Mutual Exclusion – Hardware Support *72*
2.10.6 Monitors *79*
2.10.7. Classical IPC (Synchronization) Problems *81*
2.11. DEADLOCK: INTRODUCTION 88
2.12. WHAT IS DEADLOCK? 88
2.12.1. Starvation vs. Deadlock *89*
2.13. DEADLOCK CHARACTERIZATION 89
2.14. RESOURCE-ALLOCATION GRAPH (MODELING DEADLOCK) 90
2.15. METHOD FOR HANDLING DEADLOCK //DETECTION 92
2.15.1. Deadlock Prevention *93*
2.15.2. Deadlock Avoidance *95*
2.15.3. Deadlock Detection *99*
2.15.4. Detection-Algorithm Usage *100*
2.15.5. Recovery from Deadlock *100*

SUMMARY 102

MEMORY MANAGEMENT 104
MEMORY MANAGEMENT 104

DYNAMIC LOADING 105
MEMORY MANAGEMENT BASICS 106
3.3.1 Monoprogramming Model 106
3.3.2. Address Binding (Relocation) 107
3.3.3. Logical versus Physical Address Space 107
MEMORY MANAGEMENT UNIT (MMU) 108
3.1.1. Memory Allocation Techniques 109
SWAPPING 115
3.1.2. Storage placement Strategies (Dynamic Partitioning Placement Algorithm) 116
3.5.2. Paging 118
3.5.3. Segmentation 125
VIRTUAL MEMORY 128
3.6.1. Demand Paging 129
3.6.2. Page Replacement 131
3.6.3. Page Replacement Algorithm 132
3.6.4. Belady's Anomaly 135
3.6.5. Thrashing 136

SUMMARY 137

I/O MANAGEMENT 139
4.1. I/O HARDWARE 139
4.1.1. Direct I/O instructions 140
4.1.2. Memory-mapped I/O 140
4.2. POLLING 140
4.3. INTERRUPTS 140
4.4. I/O DEVICES 142
4.5. WAYS TO INPUT/OUTPUT 142
4.5.1. Programmed I/O 143
4.5.2. INTERRUPT DRIVEN I/O 144
4.5.3. Direct Memory Access (DMA) 144
4.6. APPLICATION I/O INTERFACED 145
4.6.1. Clocks and Timers 146
4.6.2. Blocking and Non-blocking I/O 147
4.7. KERNEL I/O SUBSYSTEM 147
4.7.1. Scheduling 147
4.7.2. Buffering 148
4.7.3. Caching 148
4.7.4. Spooling and Device Reservation 148
4.7.5. Error Handling 148
4.7.6. Device drivers 148
4.7.7. Vectored I/O 149
4.7.8. I/O Protection 149
4.8. KERNEL DATA STRUCTURES 149
4.9. POWER MANAGEMENT 150
4.10 I/O REQUESTS TO HARDWARE OPERATIONS 150
4.10.1 Life Cycle of Blocking read request 151
4.11 STREAMS 152
4.12. DEVICE FUNCTIONALITY 153

4.13. MASS-STORAGE DEVICE 154
4.13.1. Disk Structure *154*
4.13.2. Disk Attachment *154*
4.13.3. Network-Attached Storage *154*
4.13.4. Storage Area Network *155*
4.13.5. Disk Scheduling *155*
4.13.6. Disk Management *159*
4.13.7. Swap Space Management *162*
4.13.8. Stable Storage Implementation *162*

SUMMARY 166

FILE SYSTEMS 167
5.1. FILE CONCEPT 167
5.1.1. File Structure *167*
5.1.2. File Attributes *167*
5.1.3. File Operations *168*
5.1.4. File Types – Name, Extension *168*
5.1.5. File Management Systems *169*
5.1.6. File System Architecture *169*
5.1.7. File-System Mounting *170*
5.1.8. Access Methods *170*
5.2. DIRECTORY STRUCTURE 171
5.2.1. Information in a Device Directory *172*
5.2.2. Operations Performed on Directory *172*
5.2.3. Organize the Directory (Logically) to Obtain *172*
5.2.4. Acyclic graph directories *175*
5.2.5 File Sharing *176*
5.2.6. Remote File Systems *176*
5.2.7. Failure Modes *176*
5.2.8. Consistency Semantics *176*
5.2.9. Protection *177*
5.3. FILE SYSTEM IMPLEMENTATION 177
5.3.1. Partitions and Mounting *178*
5.3.2. Virtual File Systems *178*
5.3.3. File-System Structure *179*
5.3.4. Layer File System *179*
5.4. FILE ALLOCATION METHODS 180
5.4.1. Contiguous Allocation *180*
5.4.2. Linked Allocation *181*
5.4.3. Indexed Allocation *183*
5.4.4. Combined Scheme: UNIX UFS *184*
5.5. FREE-SPACE MANAGEMENT 184
5.5.1. Bit Vector *185*
5.5.2. Linked List *185*
5.5.3. Grouping *185*
5.5.4. Counting *186*
5.6. EFFICIENCY AND PERFORMANCE 186
5.7. PAGE CACHE 186

5.8. RECOVERY 187
5.8.1. Log Structured File Systems 187
5.9. DIRECTORY IMPLEMENTATION 187
5.9.1. Linear list 188
5.9.2. Hash table 188
5.10. BACKUP AND RESTORE 188
SUMMARY 188
REFERENCES 190

List of Figures

Figure 1: Conceptual view of a computer system 2
Figure 2: Computer organization system 6
Figure 3: Storage hierarchy 7
Figure 4: Symmetric multiprocessor architecture 8
Figure 5: Dual core design 8
Figure 6: Transition from user mode to kernel mode 9
Figure 7: Client server system 10
Figure 8: Memory layout for a simple batch system 11
Figure 9: Memory layout for a multiprogramming system 13
Figure 10.CPU usage in uniprogramming 13
Figure 11: CPU usage in multiprogramming 14
Figure 12: Example of system call 19
Figure 13: API, system call and OS relationship 20
Figure 14: API, system call and OS relationship 21
Figure 15: Parameter passing via table 21
Figure 16: System structure 22
Figure 17: System structure 23
Figure 18: Monolithic structure 23
Figure 19.Layered structure-I 24
Figure 20: Layered structured-II 24
Figure 21: Layered structure-III 24
Figure 22: Microkernel structure 24
Figure 23: Client Server structure 25
Figure 24: Virtual machines 25
Figure 25: Modules Structure 26
Figure 26: Diagram for Process State 31
Figure 27: Process Control Block 32
Figure 28: Active process in Linux 33
Figure 29: CPU switching between processes 33
Figure 30: Queuing Diagram 34
Figure 31: Queueing diagram with medium term scheduler 36
Figure 32: Fork () Process 38
Figure 33: Communication Model (a) Message Passing b) Shared Memory 40
Figure 34: Single and multithreaded process. 43
Figure 35: Multithreaded Server Architecture 43
Figure 36: Parallel Execution on a Multi Core system 44
Figure 37: User Level Thread 45
Figure 38: Kernel Level thread 45
Figure 39: Hybrid Thread 46
Figure 40: Many-to-Many Model 46
Figure 41: Many to one model. 47
Figure 42: One to One Thread 47
Figure 43: Alternating Sequence of CPU and I/O bursts 50
Figure 44: A scheduling algorithm with four Priority classes 57
Figure 45: How a smaller time quantum increases context switches 60
Figure 46: Turnaround time varies with quantum time 60
Figure 47: Multi Level Queue Scheduling 61
Figure 48: Example of Multilevel feedback queue 62
Figure 49: Mutual Exclusion Using Critical Regions 68
Figure 50: The producer-consumer problem with a fatal race condition 77
Figure 51: Consumer 82

Figure 52: A solution to the dining philosophers' problem. *84*
Figure 53: A solution to the readers and writers problem. *85*
Figure 54: A solution to the sleeping barber problem. *87*
Figure 55: Resource Allocation Graph with Deadlock *91*
Figure 56: Resource Allocation Graph with cycle but no deadlock *92*
Figure 57: Safe, Unsafe & Deadlock State *95*
Figure 58: Demonstration that the figure in (a) is safe state *96*
Figure 59: demonstration that the figure in (b) is not safe *96*
Figure 60: (a): Resource Allocation Graph Fig (b): Corresponding Wait for Graph *99*
Figure 61: A base and limit register define a logical address space *104*
Figure 62: Memory management Unit *108*
Figure 63: Dynamic relocation using relocation register *108*
Figure 64: Memory protection using relocation register *108*
Figure 65: Example of fixed partitioning of a 64-Mbyte memory *110*
Figure 66: (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue *111*
Figure 67: Effects of dynamic programming *112*
Figure 68: Coalescing *113*
Figure 69: Compaction *113*
Figure 70: Swapping of two processes using a disk as a blocking store *115*
Figure 71: Paging Hardware *119*
Figure 72: Paging Model *120*
Figure 73: 32-byte memory and 4-byte pages *121*
Figure 74: a) Before allocation b) After Allocation *121*
Figure 75: (a) A 32-bit address with two page table fields. *121*
Figure 76: Paging Hardware with TLB *122*
Figure 77: Hashed Page Tables *125*
Figure 78: Inverted Page Table *125*
Figure 79: Logical view of segmentation *126*
Figure 80: Segmentation hardware *127*
Figure 81: Example of Segmentation *127*
Figure 82: Diagram showing virtual memory that is larger than physical memory. *129*
Figure 83: Transfer of a paged memory to continuous disk space *129*
Figure 84: The Picture When All Pages Are Not In Memory *130*
Figure 85: The Picture When All Pages Are Not In Memory *131*
Figure 86: Page replacement *131*
Figure 87: FIFO page replacement algorithm *132*
Figure 88: Optimal page replacement algorithm *133*
Figure 89: LRU page replacement algorithm *133*
Figure 90: Belady's anomaly. (a) FIFO with three page frames. (b) FIFO with four page frames. The P's show which page references cause page faults. *136*
Figure 91: Thrashing *136*
Figure 92: A typical PC bus structure *140*
Figure 93: Interrupts driven I/O Cycle *141*
Figure 94: Steps in printing a string *143*
Figure 95: Six Step Process to Perform DMA Transfer *145*
Figure 96: Two I/O model a) Synchronous b) Asynchronous *147*
Figure 97: Kernel I/O Subsystem *147*
Figure 98: Device Status Table *149*
Figure 99: Use of a System Call to Perform I/O: *149*
Figure 100: UNIX I/O Kernel Structure *150*
Figure 101: Life cycle of an I/O Request *151*
Figure 102: The STREAM Structure *152*

Figure 103: Intercomputer Communication 152
Figure 104: Networked Attached Storage 155
Figure 105: Storage Area Network 155
Figure 106: FIFO disk scheduling algorithm 156
Figure 107: SSTF disk scheduling Algorithm 157
Figure 108: SCAN disk scheduling algorithm 158
Figure 109: C-SCAN disk scheduling algorithm 158
Figure 110: C-LOOK disk scheduling Algorithm 159
Figure 111: Bad Block Recovery (a) A disk track with a bad sector. (Sector 7 is bad) (b) Substituting a spare for the bad sector. (Sector Sparing or Forwarding (c) Shifting all the sectors to bypass the bad one. (Sector Slipping) 161
Figure 112: Interleaving (a) No Interleaving (b) Single Interleaving (c) Double Interleaving 162
Figure 113: Sequential access of file 171
Figure 114: Simulation of Sequential Access on a Direct-access File 171
Figure 115: Example of Index and Relative Files 171
Figure 116: A typical File system organization 172
Figure 117: Single Level Directory 173
Figure 118: Two Level Directoory 174
Figure 119: Tree structured directories 175
Figure 120: Acyclic graph directories 176
Figure 121: Opening and Reading a file (a) refers to opening a file (b) refers to reading a file 178
Figure 122: Systematic view of virtual file system 179
Figure 123: Contiguous Allocation 180
Figure 124: Linked Allocation 182
Figure 125: File Allocation Table 183
Figure 126: Index Allocation 183
Figure 127: An i-node with three levels of indirect blocks 184
Figure 128: Combined Scheme: UNIX UFS 184
Figure 129: Linked List scheme 185
Figure 130: I/O without unified buffer 187
Figure 131: I/O using unified buffer 187

List of Tables

Table 1: History of an operating System *4*
Table 2: Difference between Program and Process *30*
Table 3: Comparison of Short Term, Medium Term and Long Term Scheduler *36*
Table 4 : Process vs Thread *44*
Table 5: Difference Between User Level Threads and Kernel Level threads *47*
Table 6: Preemptive vs Non-preemptive scheduling *51*
Table 7: Fixed partition vs Variable partition *114*
Table 8: Paging vs Segmentation *128*
Table 9: File Types *168*

CHAPTER 1

OPERATING SYSTEM TYPES AND STRUCTURE

1.1 Introduction

An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.

An operating system is software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper Operation of the system.

1.2 Operating System

1.2.1 Definition of Operating System:

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

A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being applications programs.

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.

1.2.2 Functions of Operating System

Operating system performs three functions:

1. ***Convenience****:* An OS makes a computer more convenient to use.
2. ***Efficiency***: An OS allows the computer system resources to be used in an efficient manner.
3. ***Ability to Evolve***: An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system functions without at the same time interfering with service.

1.2.3 Operating System as User Interface

Every general-purpose computer consists of the hardware, operating system, system programs, and application programs. The hardware consists of memory, CPU, ALU, I/O devices, peripheral device and storage device. System program consists of compilers, loaders, and editors, OS etc. The application program consists of business program, database program. The figure shows the conceptual view of a computer system

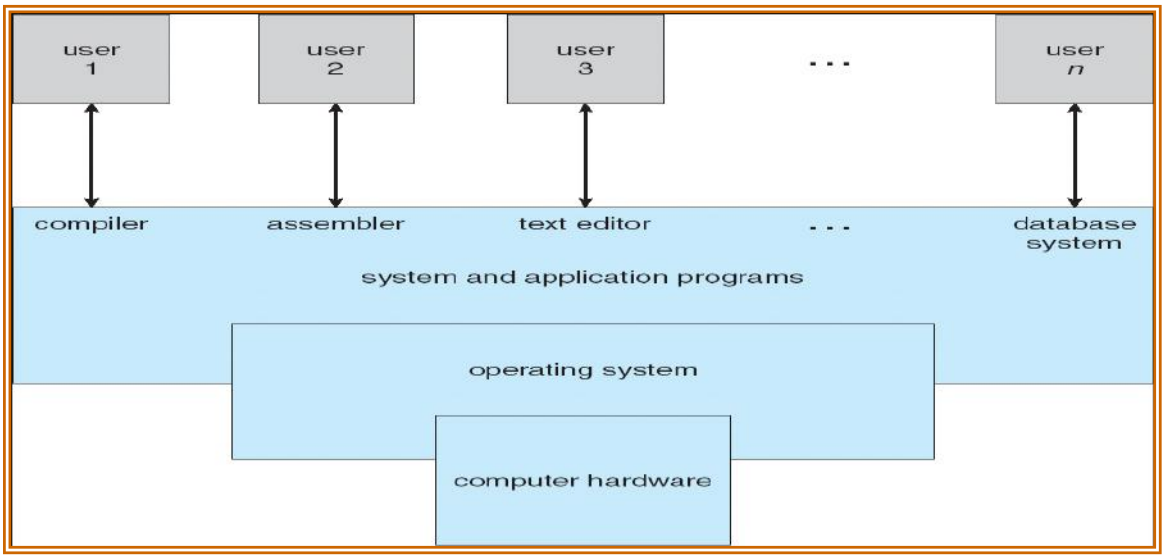


Figure 1: Conceptual view of a computer system

Every computer must have an operating system to run other programs. The operating system and coordinates the use of the hardware among the various system programs and application program for a various users. It simply provides an environment within which other programs can do useful work.

The operating system is a set of special programs that run on a computer system that allow it to work properly. It performs basic tasks such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling a peripheral device.

OS is designed to serve two basic purposes:

- It controls the allocation and use of the computing system‘s resources among the various user and tasks.
- It provides an interface between the computer hardware and the programmer that simplifies and makes feasible for coding, creation, debugging of application programs.

The operating system must support the following tasks. The tasks are:

- Provides the facilities to create, modification of program and data files using and editor.
- Access to the compiler for translating the user program from high level language to machine language.
- Provide a loader program to move the compiled program code to the computer‘s memory for execution.
- Provide routines that handle the details of I/O programming.

1.3 I/O System Management

The module that keeps track of the status of devices is called the I/O traffic controller. Each I/O device has a device handler that resides in a separate process associated with that device.

The I/O subsystem consists of

- A memory management component that includes buffering, caching and spooling.
- A general device driver interface.
- Drivers for specific hardware devices.

1.4 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. At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of ones and zeros(machine language), place them into the memory of the machine.

1.5 Compiler

The high level languages – examples are FORTRAN, COBOL, ALGOL and PL/I are processed by compilers and interpreters. A compiler is a program that accepts a source program in a high-level language and produces a corresponding object program. An interpreter is a program that appears to execute a source program as if it was machine language. The same name (FORTRAN, COBOL etc.) is often used to designate both a compiler and its associated language.

1.6 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. Loader is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary device and a loader is placed in core. The loader places into memory the machine language version of the user‘s program and transfers control to it. Since the loader program is much smaller than the assembler, this makes more cores available to user‘s program.

1.7. History of Operating System

Operating systems have been evolving through the years. Following table shows the history of OS.

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

Table 1: History of an operating System

1.8 Introduction to Operating System

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. The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins.

We can view an operating system from several vantage points. One view focuses on the services that the system provides, another, on the interface that it makes available to users and programmers; a third, on its components and their interconnections.

1.8.1. Operating System Services

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 are difficult than other operating system. Operating system makes the programming task easier.

The common service provided by the operating system is listed below.

1. Program execution
2. I/O operation
3. File system manipulation
4. Communications
5. Error detection

1. ***Program execution:*** Operating system loads a program into memory and executes the program. The program must be able to end its execution, either normally or abnormally.
2. ***I/O Operation:*** I/O means any file or any specific I/O device. Program may require any I/O device while running. So operating system must provide the required I/O.
3. ***File system manipulation:*** Program needs to read a file or write a file. The operating system gives the permission to the program for operation on file.
4. ***Communication***: Data transfer between two processes is required for some time. The both processes are on the one computer or on different computer but connected through computer network. Communication may be implemented by two methods:
    a. Shared memory
    b. Message passing.
5. ***Error detection:*** error may occur in CPU, in I/O devices or in the memory hardware. The operating system constantly needs to be aware of possible errors. It should take the appropriate action to ensure correct and consistent computing.

Operating system with multiple users provides following services.

1.Resource Allocation
2.Accounting

3.Protection

1.Resource Allocation:

- If there are more than one user or jobs running at the same time, then resources must be allocated to each of them. Operating system manages different types of resources require special allocation code, i.e. main memory, CPU cycles and file storage.
- There are some resources which require only general request and release code. For allocating CPU, CPU scheduling algorithms are used for better utilization of CPU. CPU scheduling algorithms are used for better utilization of CPU. CPU scheduling routines consider the speed of the CPU, number of available registers and other required factors.

2.Accounting:

- Logs of each user must be kept. It is also necessary to keep record of which user how much and what kinds of computer resources. This log is used for accounting purposes.
- The accounting data may be used for statistics or for the billing. It also used to improve system efficiency.

3.Protection:

- Protection involves ensuring that all access to system resources is controlled.
- Security starts with each user having to authenticate to the system, usually by means of a password. External I/O devices must be also protected from invalid access attempts.
- In protection, all the access to the resources is controlled. In multiprocessor environment, it is possible that, one process to interface with the other, orwith the operating system, so protection is required.

1.8.2. Operating System Components

Modern operating systems share the goal of supporting the system components. The system components are:

1. Process Management
2. Main Memory Management
3. File Management
4. Secondary Storage Management
5. I/O System Management
6. Networking
7. Protection System
8. Command Interpreter System

1.8.3. Computer Startup

The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate and load into memory the operating system kernel. The operating system then starts executing the first process, such as "init," and waits for some event to occur.

1.8.4. Computer System Organization

- One or more CPUs, device controllers connect through common bus providing access to shared memory
- Concurrent execution of CPUs and devices competing for memory cycles

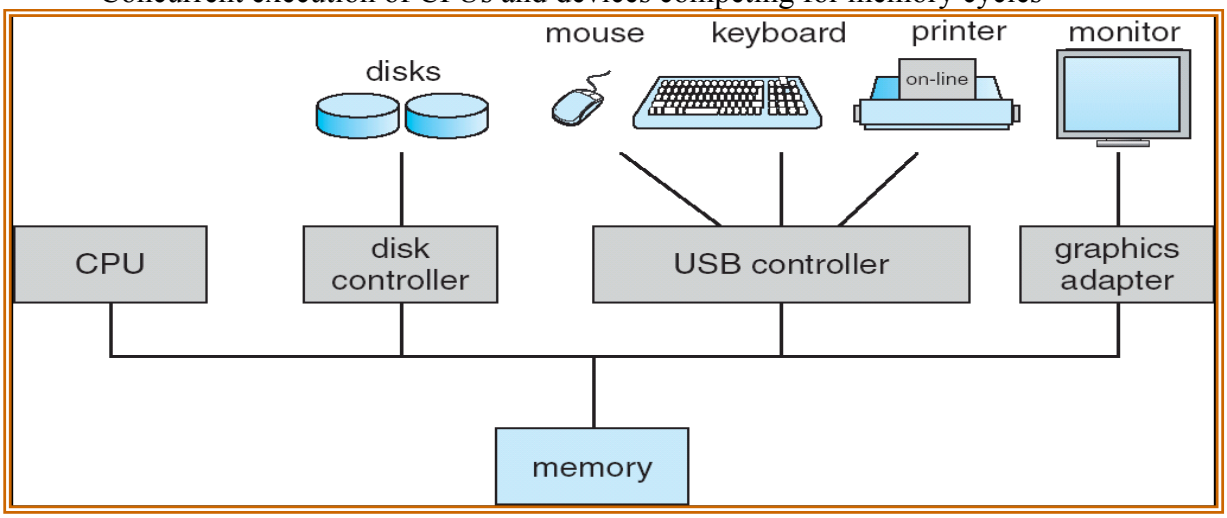


Figure 2: Computer organization system

1.8.5. Computer-System Operation

- I/O devices and CPU can execute concurrently
- Each device controller is in charge of a device type
- Each device controller has a local buffer

I/O operations

- I/O is from the device to the local buffer
- Device controller informs CPU by causing an interrupt
- CPU moves data between local buffers and main memory

1.8.6. Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine
- through interrupt vector, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt
- A trap is a software-generated interrupt caused either by an error or a user request
- An interrupt driven OS

1.8.7. I/O Structure

- **Synchronous**: After I/O starts, control returns to user program only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request at a time, no simultaneous I/O processing

- **Asynchronous:** After I/O starts, control returns to user program without waiting for I/O completion
- ***System call*** – request to the OS to allow user to wait for I/O completion
- ***Device-status table*** for each I/O device: type, address, and state
    - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt

1.8.8. Direct Memory Access Structure

- Used for high-speed I/O devices
    - able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

1.8.9. Storage Structure

- **Main memory** – the only large storage media that the CPU can access directly
- **Secondary storage** – extension of main memory that provides large nonvolatile storage capacity
    - **Magnetic disks** – rigid metal or glass platters covered with magnetic recording material
    - Disk surface is logically divided into ***tracks***, which are subdivided into ***sectors***
    - The ***disk controller*** determines the logical interaction between the device and the computer.

1.8.10. Storage Hierarchy

- Storage systems organized in hierarchy.
    - Speed
    - Cost
    - Volatility
- ***Caching*** – copying information into faster storage system
    - main memory as a last *cache* for secondary storage

According to speed and cost.

- The higher levels are expensive, but they are fast.
- As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.
- This trade-off is reasonable.
- The top four levels of memory may be constructed using semiconductor memory.

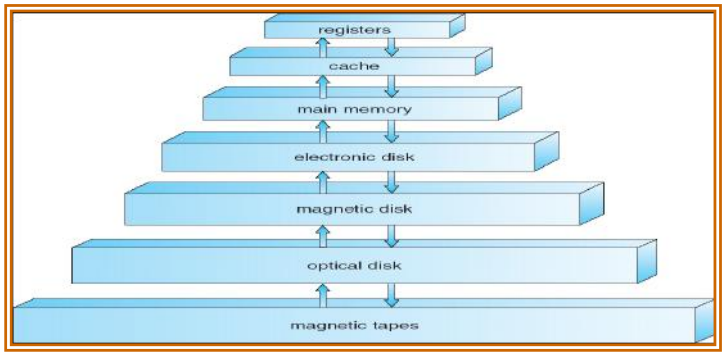


Figure 3: Storage hierarchy

1.9. Computer-System Architecture

- Single processor systems

- General-purpose and special-purpose processors
- Multiprocessor systems growing in use and importance
  - Also known as parallel systems, tightly-coupled systems
- Advantages include
  - Increased throughput
  - Economy of scale
  - Increased reliability – graceful degradation or fault tolerance
- Two types
  - Asymmetric Multiprocessing
  - Symmetric Multiprocessing (SMP)

1.10 Symmetric Multiprocessing Architecture

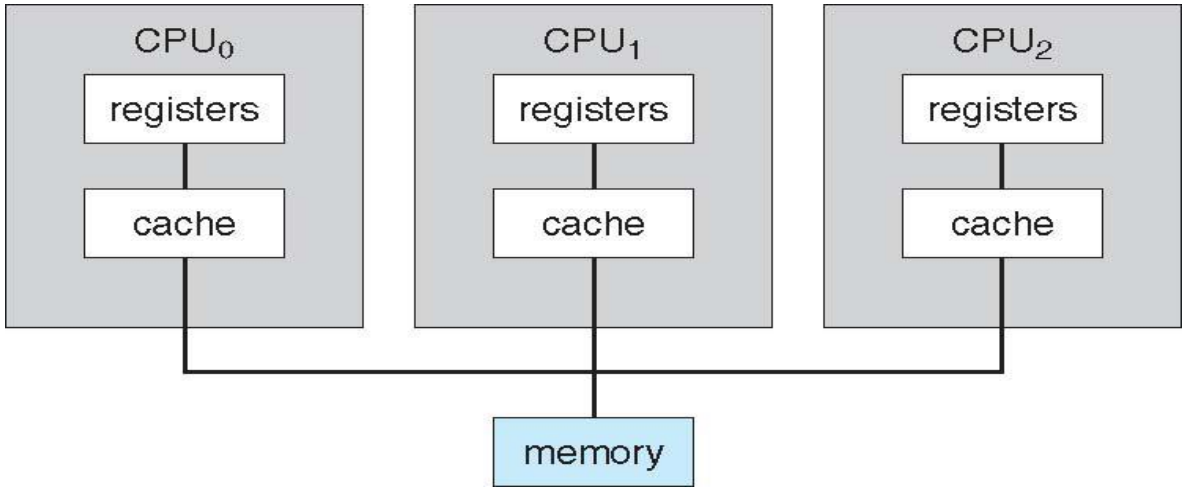


Figure 4: Symmetric multiprocessor architecture

1.11. A Dual-Core Design

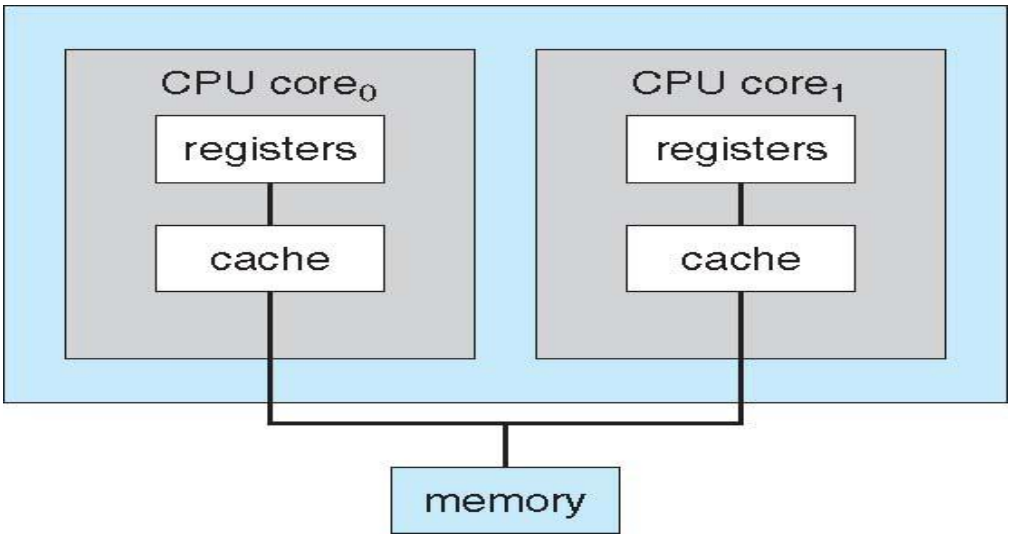


Figure 5: Dual core design

1.12. Clustered Systems

- Like multiprocessor systems, but multiple systems working together
- Usually sharing storage via a storage-area network (SAN)

- Provides a high-availability service which survives failures
- Asymmetric clustering has one machine in hot-standby mode
- Symmetric clustering has multiple nodes running applications, monitoring each other
- Some clusters are for high-performance computing (HPC)
- Applications must be written to use parallelization

1.13. Operating-System Operations

- Interrupt driven by hardware
- Software error or request creates exception or trap
  - Division by zero, request for OS service
  - Other process problems include infinite loop, processes modifying each other or the OS
- Dual-mode operation allows OS to protect itself and other components
  - User mode and kernel mode
  - Mode bit provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as privileged, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

1.13.1. Transition from User to Kernel Mode

- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - OS decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time

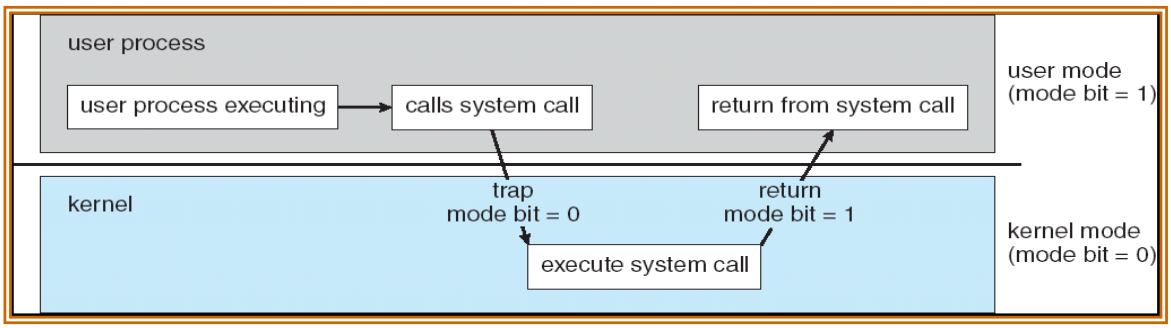


Figure 6: Transition from user mode to kernel mode

1.14. I/O Subsystem

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

- Drivers for specific hardware devices

1.15. Protection and Security

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

1.16. Computing Environments

- Traditional computer
  - Blurring over time
  - Office environment
    - PCs connected to a network, terminals attached to mainframe or minicomputers providing batch and timesharing
    - Now portals allowing networked and remote systems access to same resources
  - Home networks
  - Used to be single system, then modems
  - Now firewalled, networked
- Client-Server Computing
  - Many systems now servers, responding to requests generated by clients
  - Compute-server provides an interface for clients to request services (i.e. database)
  - File-server provides interface for clients to store and retrieve files

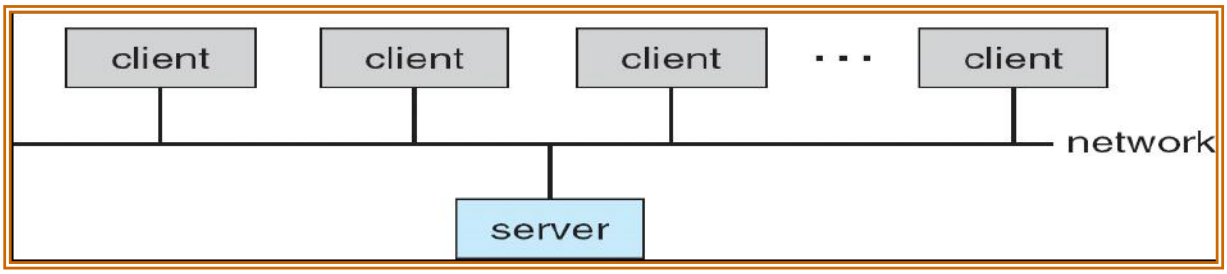


Figure 7: Client server system

1.17. Peer-to-Peer Computing

- Another model of distributed system
- P2P does not distinguish clients and servers
  - All nodes are considered peers
  - May each act as client, server or both
  - Node must join P2P network
  - Registers its service with central lookup service on network, or

- Broadcast request for service and respond to requests for service via discovery protocol
- Examples include Napster and Gnutella

1.18. Web-Based Computing

- Web has become ubiquitous(Everywhere)
  - PCs most powerful devices
  - More devices becoming networked to allow web access
- New category of devices to manage web traffic among similar servers: load balancers
- Use of operating systems like Windows 95, web clients, have evolved into Linux and Windows XP, which can be clients and servers

1.19. Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary closed-source
- Counter to the copy protection and Digital Rights Management (DRM) movement
- Started by Free Software Foundation (FSF), which has "copyleft" GNU Public License (GPL)
- Examples include GNU/Linux, BSD UNIX (including core of Mac OS X), and Sun Solaris

1.20. Batch System

Some computer systems only did one thing at a time. They had a list of the computer system may be dedicated to a single program until its completion, or they may be dynamically reassigned among a collection of active programs in different stages of execution.

- Batch operating system is one where programs and data are collected together in a batch before processing starts. A job is predefined sequence of commands, programs and data that are combined in to a single unit called job.
- Fig. 2.1 shows the memory layout for a simple batch system. Memory management in batch system is very simple. Memory is usually divided into two areas: Operating system and user program area.

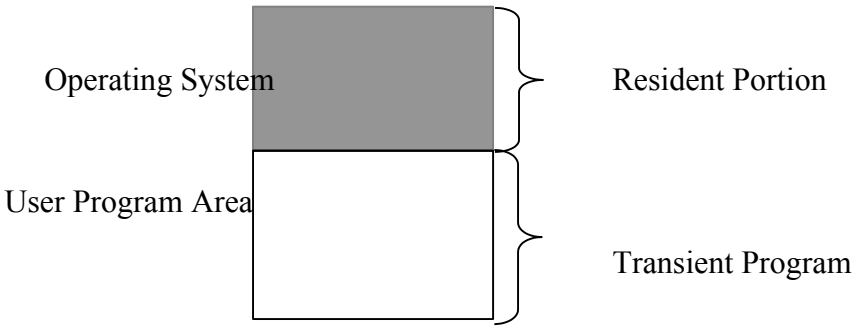


Figure 8: Memory layout for a simple batch system

- Scheduling is also simple in batch system. Jobs are processed in the order of submission i.e. first come first served fashion.

- When job completed execution, its memory is releases and the output for the job gets copied into an output spool for later printing.
- Batch system often provides simple forms of file management. Access to file is serial. Batch systems do not require any time critical device management.
- Batch systems are inconvenient for users because users cannot interact with their jobs to fix problems. There may also be long turnaround times. Example of this system id generating monthly bank statement.

Advantages of Batch System

- Increased performance since it was possible for job to start as soon as the previous job finished.

Disadvantages of Batch System

- Turnaround time can be large from user standpoint. Difficult to debug program.
- A job could enter an infinite loop.
- A job could corrupt the monitor, thus affecting pending jobs.
- Due to lack of protection scheme, one batch job can affect pending jobs.

1.21. Time Sharing Systems

- Multi-programmed batched systems provide an environment where the various system resources (for example, CPU, memory, peripheral devices) are utilized effectively.
- 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.
- An interactive, or hands-on, computer system provides on-line communication between the user and the system. The user gives instructions to the operating system or to a program directly, and receives an immediate response. Usually, a keyboard is used to provide input, and a display screen (such as a cathode-ray tube (CRT) or monitor) is used to provide output.
- If users are to be able to access both data and code conveniently, an on-line file system must be available. A file is a collection of related information defined by its creator. Batch systems are appropriate for executing large jobs that need little interaction.
- Time-sharing systems were developed to provide interactive use of a computer system at a reasonable cost. A time-shared operating system uses CPU scheduling and multiprogramming 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 that is loaded into memory and is executing is commonly referred to as 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 is to a display for the user and input is from a user keyboard. Since interactive I/O typically runs at people speeds, it may take a long time to complete.
- A time-shared operating system allows the 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 she has her own computer, whereas actually one computer is being shared among many users.

- Time-sharing operating systems are even more complex than are multi- programmed operating systems. As in multiprogramming, several jobs must be kept simultaneously in memory, which requires some form of memory management and protection.

1.22. Multiprogramming

When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system. Multiprogramming assumes a single processor that is being shared. It increases CPU utilization by organizing jobs so that the CPU always has one to execute.

Figure 9 shows the memory layout for a multiprogramming system.

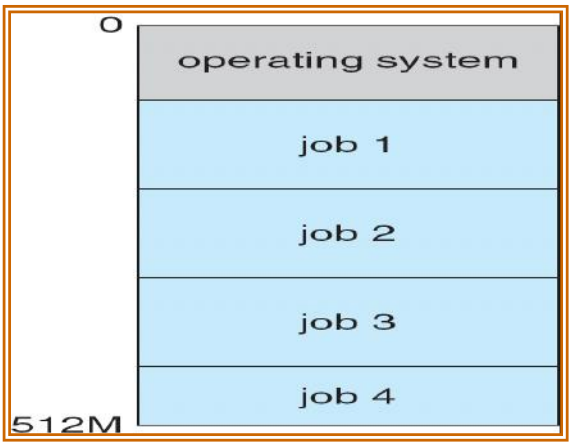


Figure 9: Memory layout for a multiprogramming system

- The operating system keeps several jobs in memory at a time. This set of jobs is a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the job in the memory.
- Multiprogrammed system provide an environment in which the various system resources are utilized effectively, but they do not provide for user interaction with the computer system.
- Jobs entering into the system are kept into the memory. Operating system picks the job and begins to execute one of the job in the memory. Having several programs in memory at the same time requires some form of memory management.
- Multiprogramming operating system monitors the state of all active programs and system resources. This ensures that the CPU is never idle unless there are no jobs.

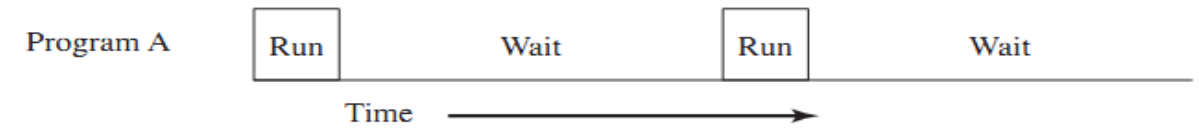


Figure 10.CPU usage in uniprogramming

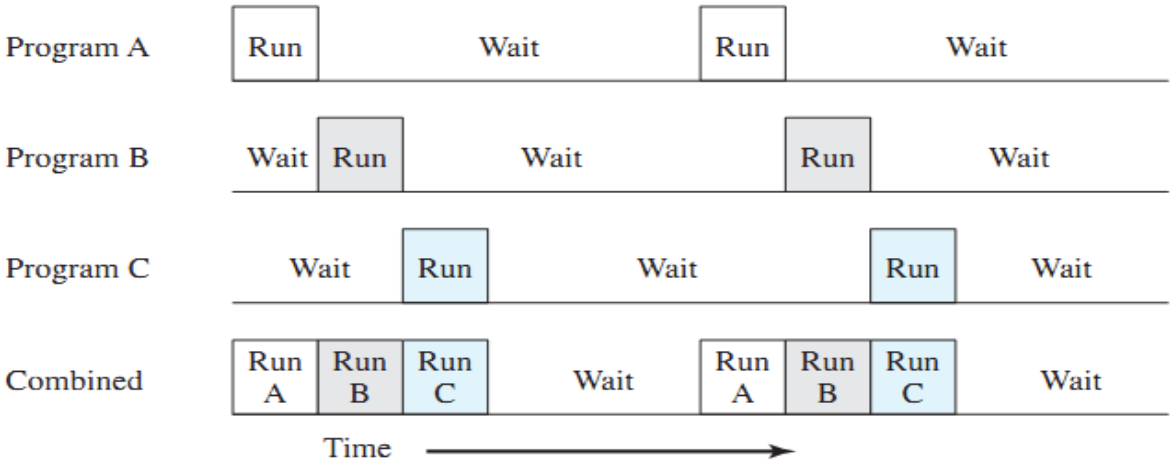


Figure 11: CPU usage in multiprogramming

Advantages

1. High CPU utilization.
2. It appears that many programs are allotted CPU almost simultaneously.

Disadvantages

1. CPU scheduling is requires.
2. To accommodate many jobs in memory, memory management is required.

1.23. Spooling

Acronym for simultaneous peripheral operations on line. Spooling refers to putting jobs in a buffer, a special area in memory or on a disk where a device can access them when it is ready.

- Spooling is useful because device access data that different rates. The buffer provides a waiting station where data can rest while the slower device catches up.
- Computer can perform I/O in parallel with computation, it becomes possible to have the computer read a deck of cards to a tape, drum or disk and to write out to a tape printer while it was computing. This process is called spooling.
- The most common spooling application is print spooling. In print spooling, documents are loaded into a buffer and then the printer pulls them off the buffer at its own rate.
- Spooling is also used for processing data at remote sites. The CPU sends the data via communications path to a remote printer. Spooling overlaps the I/O of one job with the computation of other jobs.
- One difficulty with simple batch systems is that the computer still needs to read the decks of cards before it can begin to execute the job. This means that the CPU is idle during these relatively slow operations.
- Spooling batch systems were the first and are the simplest of the multiprogramming systems.

Advantage of Spooling

1. The spooling operation uses a disk as a very large buffer.
2. Spooling is however capable of overlapping I/O operation for one job with processor operations for another job.

1.24. Parallel Systems

- Flynn's Classification:
  - Single Instruction, Single Data (SISD)
  - Single Instruction, Multiple Data (SIMD)
  - Multiple Instructions, Single Data (MISD)
  - Multiple Instructions, Multiple Data (MIMD) - multiprocessing
- many calculations are carried out simultaneously
- complex problems are divided into smaller ones, which are then solved concurrently
- Can be achieved via:
  - Multiple cores or processors within single system, or

  - Multiple computers as clusters or grids
- Multiprocessing can be:
  - Tightly Coupled (Shared Memory System)
  - Loosely Coupled (Distributed memory System)

Types:

- Asymmetric multiprocessing:
  - Each processor assigned a specific task
  - Presence of controller processor (Master-slave relationship)
- Symmetric multiprocessing:
  - Each processor performs all tasks within OS
  - All processors are peers (no mater-slave relationship)

Advantages:

- Increased Throughput
- Economy of Scale
- Increased Reliability

1.25. Real Time Systems

- Time interval required to process and respond to inputs is so small that it controls the environment.
- Real time processing is always on line
- Response time is very less
- Used when there are rigid time requirements on the operation of a processor or the flow of data
- Can be used as a control device in a dedicated application.
- Real-time OS has well-defined, fixed time constraints otherwise system will fail.
- Example:
  - Scientific experiments,
  - industrial control systems,
  - weapon systems,
  - robots,
  - home-appliance controllers,
  - Air traffic control system etc.

1.26. Distributed Systems

- A collection of computers that act, work, and appear as one large computer
- Use multiple central processors to serve multiple real time application and multiple users
- Data processing jobs are distributed among the processors accordingly to which one can perform each job most efficiently
- The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines)
- These are referred as loosely coupled systems or distributed systems
- Processors in a distributed system may vary in size and function

Advantages:

- With resource sharing facility user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.

- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

1.27. Essential Properties of the Operating System

1. ***Batch:*** Jobs with similar needs are batched together and run through the computer as a group by an operator or automatic job sequencer. Performance is increased by attempting to keep CPU and I/O devices busy at all times through buffering, off line operation, spooling and multiprogramming. A Batch system is good for executing large jobs that need little interaction, it can be submitted and picked up latter.

2. ***Time sharing:*** Uses CPU s scheduling and multiprogramming to provide economical interactive use of a system. The CPU switches rapidly from one user to another i.e. the CPU is shared between a numbers of interactive users. Instead of having a job defined by spooled card images, each program reads its next control instructions from the terminal and output is normally printed immediately on the screen.

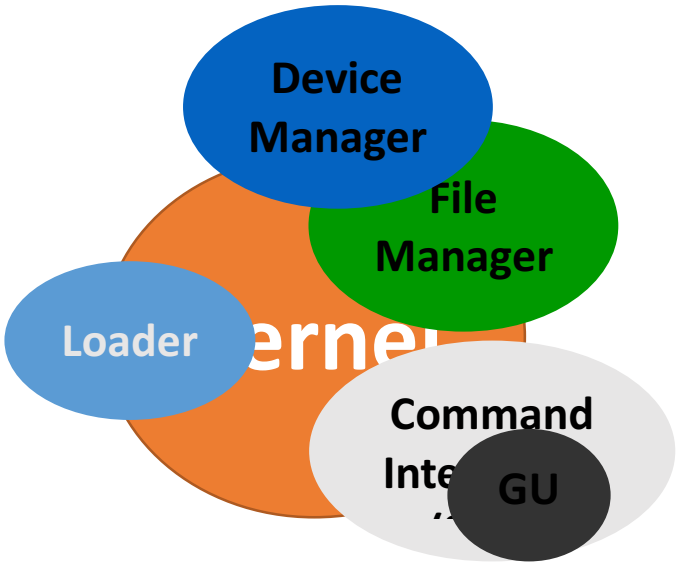
3. ***Interactive:*** User is on line with computer system and interacts with it via an interface. It is typically composed of many short transactions where the result of the next transaction may be unpredictable. Response time needs to be short since the user submits and waits for the result.

4. ***Real time system:*** Real time systems are usually dedicated, embedded systems. They typically read from and react to sensor data. The system must guarantee response to events within fixed periods of time to ensure correct performance.

5. ***Distributed:*** Distributes computation among several physical processors. The processors do not share memory or a clock. Instead, each processor has its own local memory. They communicate with each other through various communication lines.

1.28. System Components

a. Process management
b. Main memory management
c. File management
d. I/O management
e. Secondary storage management
f. Networking
g. Protection
h. Command interpreter system



1.28.1. Process Management

- A process is:
  - Active program in a system
  - Program on the run
  - Unit of work in a system

OS responsibility

- Create/delete user & system processes
- Suspend/resume processes
- Process synchronization
- Inter-process communication
- Deadlock handling

1.28.2. Main Memory Management

- All data and instruction in a system are accessed via main memory
- Main memory is addressed for referring by CPU
- The fetching of instruction and data as well as storage of data is done at main memory
- Programs are loaded into memory before execution
- The memory is made available once the program is terminated

1.28.3. File Management

- The only visible components of OS
- Actual contents (data) are stored in system in form of files
- The storage is done in physical medium
- The file provides abstraction of the physical medium to user
- Files themselves are managed using directories

1.28.4. Input-Output Management

- OS is provided with an I/O subsystem
- Contains device driver interface
- Hardware drivers are installed
- I/O memory management is done using caching, buffering and spooling (Simultaneous Peripheral Operations On Line)

1.28.5. Secondary Storage Management

- Used for long term storage of data and program
- Very huge storage space
- More free spaces are available – hence free space should be managed
- The allocation of storage locations to different files has to be managed
- The scheduling of the disk also is managed

1.28.6. Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

1.28.7. Networking

- A network connection can be **full** (i.e. all services and resources are shared) or **partial** (i.e. only few services and/or resources are shared)

- The routing of the data has to be handled
- Different connection strategies can be implemented
- The problems during connection and data sharing has to be handled
- Different protocols for networking can be used (http, ftp, TCP …)

1.28.8. Protection

- Processes has to be protected in case of multiuser system
- Controlling of resource access among multiple users, processes or programs
- Unauthorized usage of system resources should be prohibited
- Authentication and Privileges should be properly managed

1.28.9. Command Interpreter System

- Provides interface between user and OS
- Also known as Shell (in Unix System)
- User commands and self-generated commands are executed
- Commands can deal with other components too
- E.g: MS-DOS, Unix Shells

1.29. OS Services

- Services provided by OS are categorized as:
  - Services provided to users
  - Services provided to system (in case of multiple users)

1.29.1. Services Provided to Users

- Program Execution:
  - Load program into memory, run the program and end the execution
- I/O Operations:
  - I/O operations are required by the program on the run
  - Users cannot control the I/O devices directly, hence there is a need of OS
- File System manipulation:
  - Creation and deletion of files and directories
  - Reading from the file and writing into the file
- Search of files
  - Search of content within the files
- Communication:
  - Exchange of information between processes of same or networked computers.
  - Communication is done using packets of information
- Error Detection:
  - Error may occur in a system due to
- Memory error
- Power failure
  - I/O error (connection error on network, paper failure on printer …)
  - User program error (illegal memory location access error, divide by zero error …)

1.29.2. Services provided to system

- Resource allocation:
  - When multiple user or jobs, resources have to be allocated properly
  - Resources can be CPU cycle, main memory, file storage, I/O devices etc.

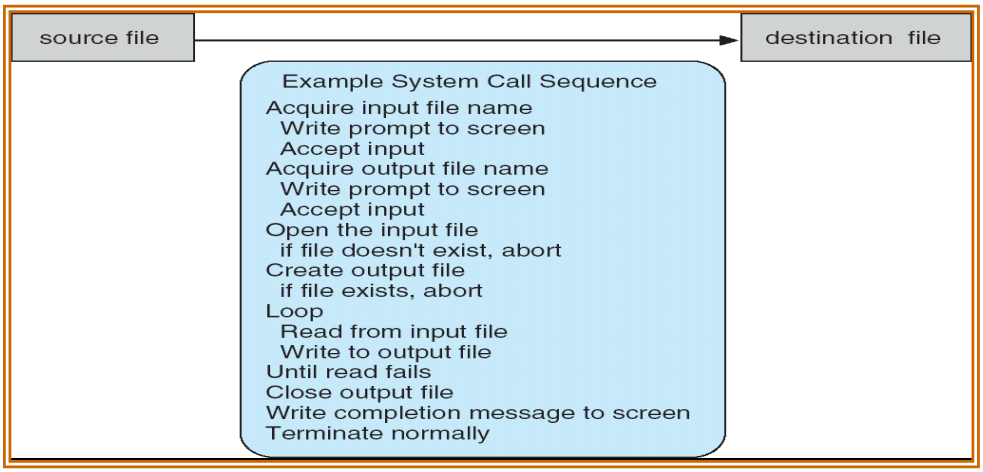
- OS is responsible for allocating the resources properly (e.g. CPU scheduling)

- Accounting:
  - OS keeps track of users on the system
  - The record keeping will be about the time and kind of resources used by each of the users
- Protection and Security:
  - *Protection m*eans to ensure that
    - all access to system resources is controlled, and
    - there is no interface between the disjoint processes executing concurrently
- *Security* means
- Authentication of user via password to gain system resources
  - Defending external I/O devices, modems, network adaptors from invalid access

1.30. System Calls

- Programming interface to the OS services
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)**
- Three most common APIs:
  - *Win32* API: Windows
  - *POSIX* API: POSIX-based systems (UNIX, Linux, and Mac OS X)
  - *Java* API: Java virtual machine (JVM)
- Why use APIs rather than system calls?
  - Portability
  - Actual system calls might be more detailed and difficult to work with

Example of System Calls

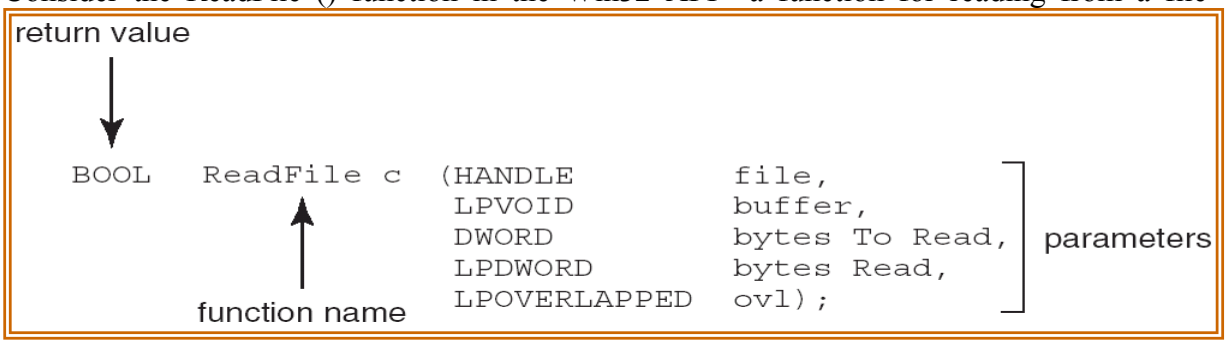


- System call sequence to copy the contents of one file to another file

Figure 12: Example of system call

Example of Standard API

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



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

1.30.1. System Call Implementation

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

API – System Call – OS Relationship

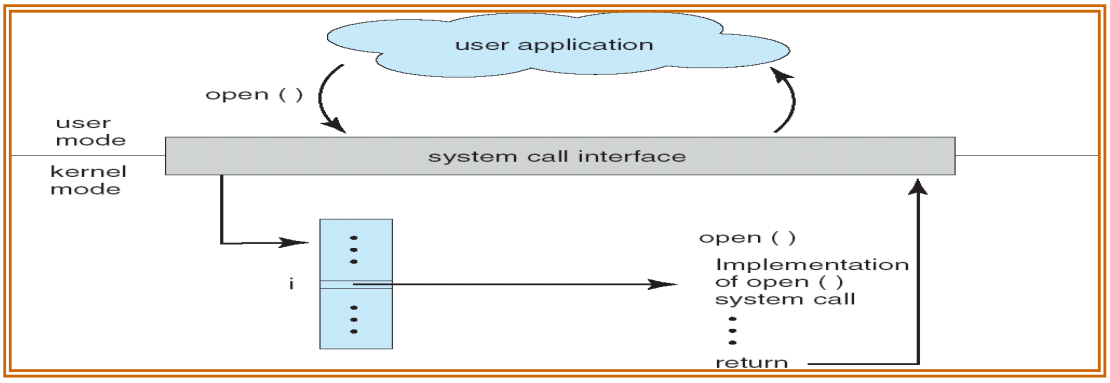


Figure 13: API, system call and OS relationship

Standard C Library Example

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

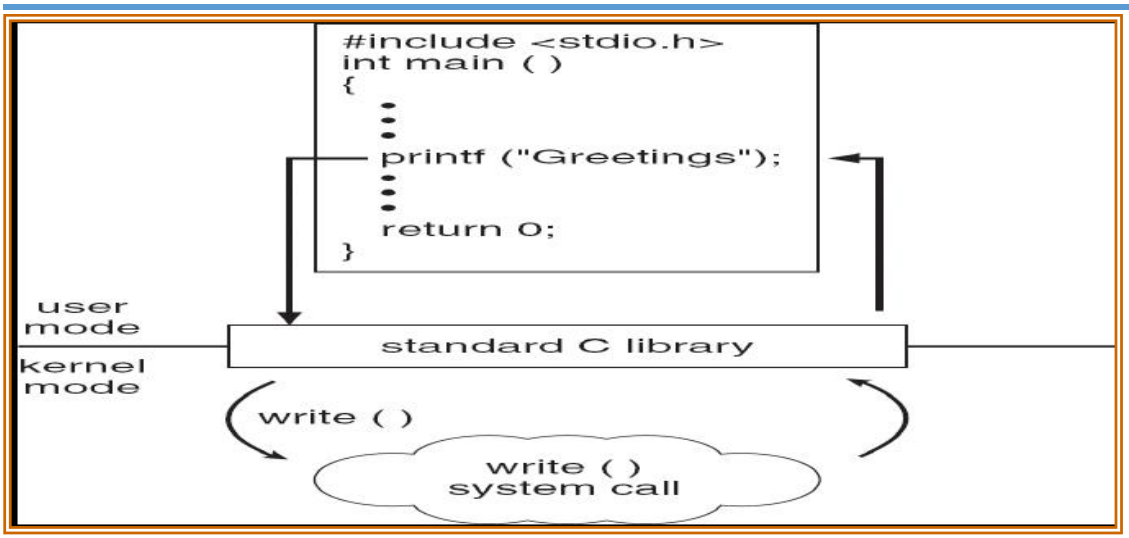


Figure 14: API, system call and OS relationship

1.30.2. System Call Parameter Passing

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

Parameter Passing via Table

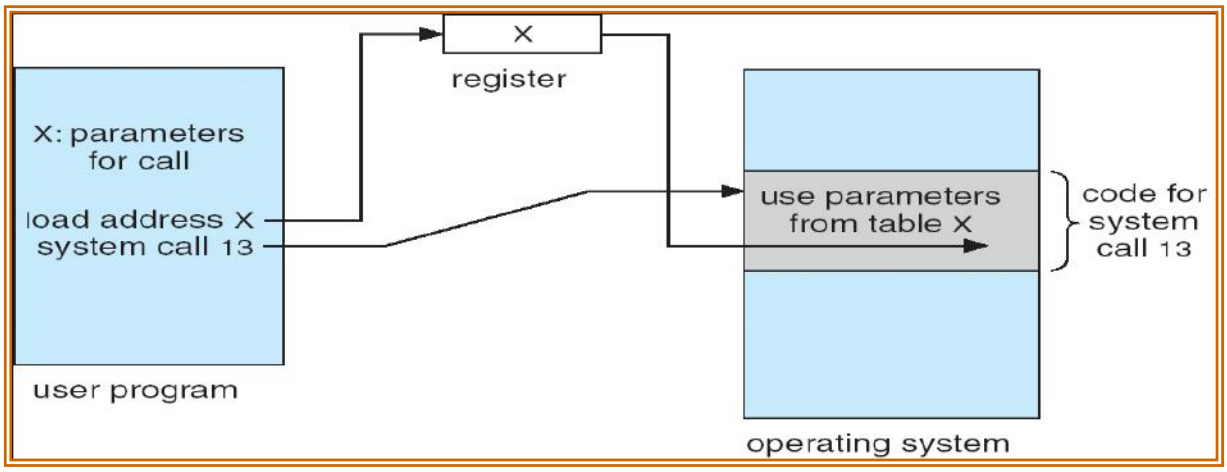


Figure 15: Parameter passing via table

Five major categories of system calls:

1. Process Control
   - End, abort, load, execute, terminate, wait, allocate memory, free memory etc

2. File management:
   - Create, delete, open, close, read write etc.
3. Device management:
   - Request, release, read, write, attach/detach devices
4. Information maintenance:
   - Get time, get date, set time, set date, get system data, set system data, get/set files, devices, attributes
5. Communication:
   - Create/delete communication connection, send/receive messages, attach/detach remote devices

1.31. System Programs

- In logical hierarchy of computer system, we can see system programs lying between OS and application programs
- System utilities, command interpreter are all system programs
- They provide convenient environment for program development and execution
- ***Categories:***

- File Management:
  - files/directories operations
- Status Information:
  - Date, time, space, users information
- Programming language support:
  - Interpreters, compilers, assembler
- Program loading and support:
  - Loaders and linkers
- Communication:
  - Remote login, email handler, file transfer

1.32. System Structure

- Interconnection of system components and their interface with kernels is the part of system structure
- System is partitioned into modules and each module well-defined portion of the system with carefully defined inputs, outputs and functions
- Some common system structures are:

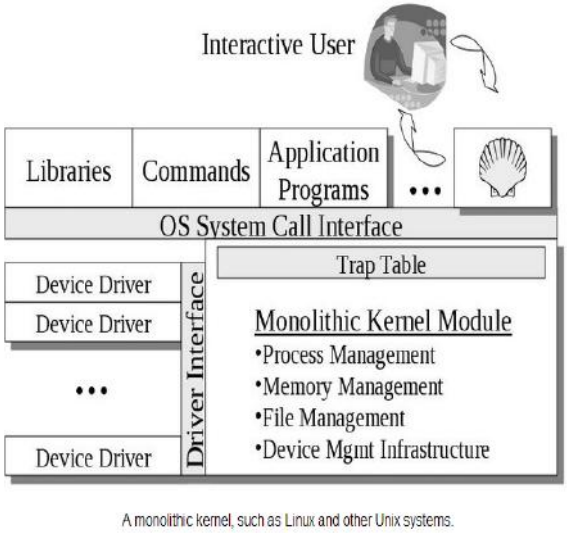


Figure 16: System structure

1.32.1. Simple Structure

- Not well defined structure
- Started as small but grew beyond the scope

- no *CPU execution mode* (user and kernel),
- So errors in applications can cause the whole system to crash.
- E.g: MS-DOS, original Unix structure

- Original UNIX OS had limited structuring because of limitation in hardware
- The UNIX OS consists of two parts:
  - **Systems programs**
  - **The kernel**
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level.

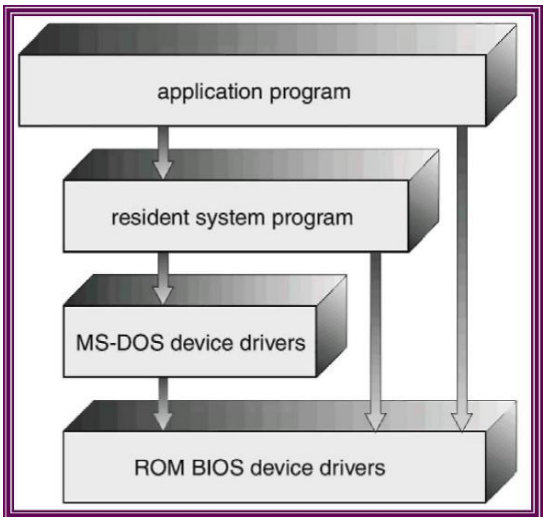


Figure 17: System structure

1.32.2. Monolithic Structure

- Here the kernel is a single large program.
- Functionality of the OS is invoked with simple function calls within the kernel
- Device drivers are loaded into the running kernel and become part of the kernel.
- System calls from user programs are kept in trap table which are executed in kernel mode switching from user mode

Three different procedures are present:

- Main procedure
  - Invokes the requested service procedure

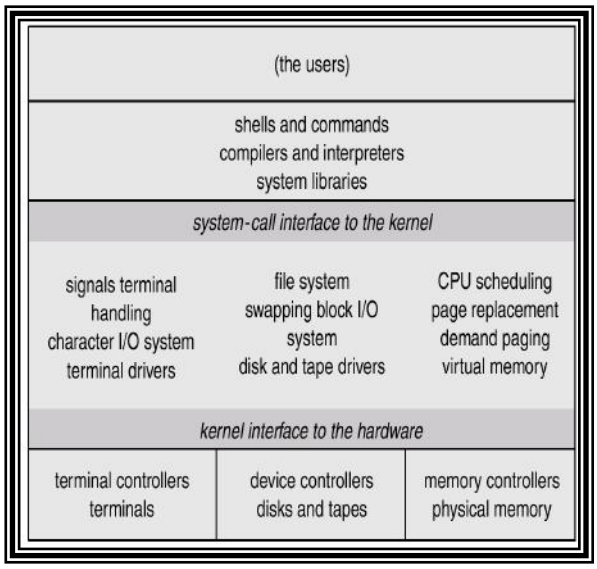
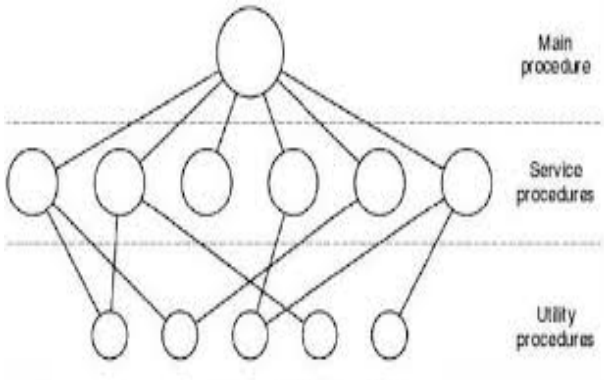


Figure 18: Monolithic structure

- Service Procedures
  - Carry out the system calls
  - For each system call, there is separate service procedure
- Utility Procedures
  - Help the service procedures
  - Such as fetching data from user programs etc.



1.32.3.Layered Structure

- Hierarchy of layers, each constructed upon another below it
- Layer 0 (hardware) to Layer N (user interface)
- Provided modularity

- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers.
- But, difficult in differentiation and less efficient

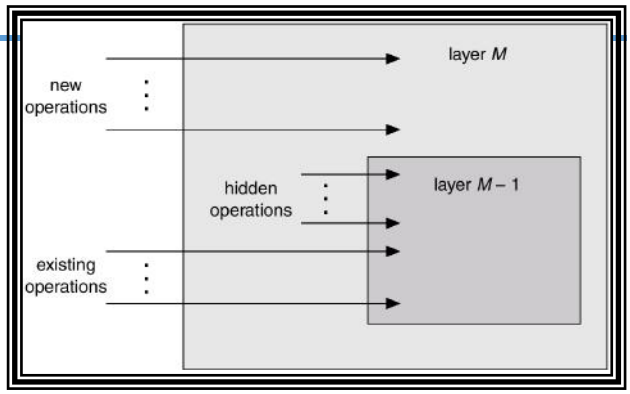


Figure 19.Layered structure-I

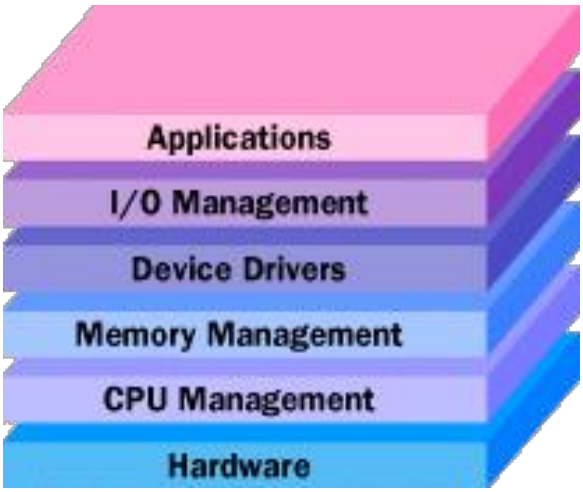


Figure 20: Layered structured-II

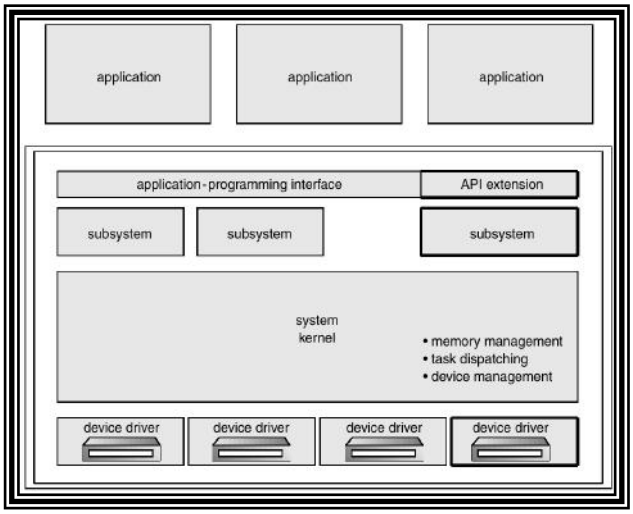


Figure 21: Layered structure-III

1.33. Microkernel Structure

- Splitting of OS into small, well defined modules
- One of the modules always resides in memory and always runs on kernel mode – *MICROKERNEL*
- Others runs as user process (device drivers, file system)
- Example:
  - OS X by apple

Benefits:

- easier to extend a microkernel
- easier to port the operating system to new architectures
- more reliable (less code is running in kernel mode)
- more secure

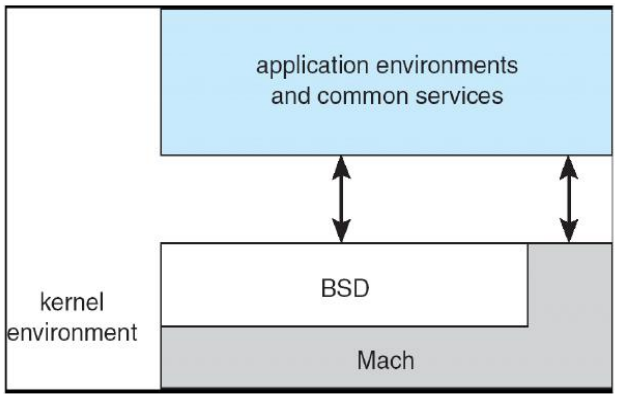


Figure 22: Microkernel structure

Mach:

- Memory management

- Remote Procedure Call (RPC)
- Inter-Process Communication (IPC)
- Message passing
- Thread Scheduling
- BSD:
- Network and file system support
- Implementation of APIs

1.34. Client Server Structure

- Two classes of processes - Server and Clients
- Communication between client and server is via message passing
- Client and server can run on different computers connected by LAN/WAN
- Servers run as user mode. Hence, no system down even if the server crashed
- Well adopted in distributed system
- E.g. Windows NT

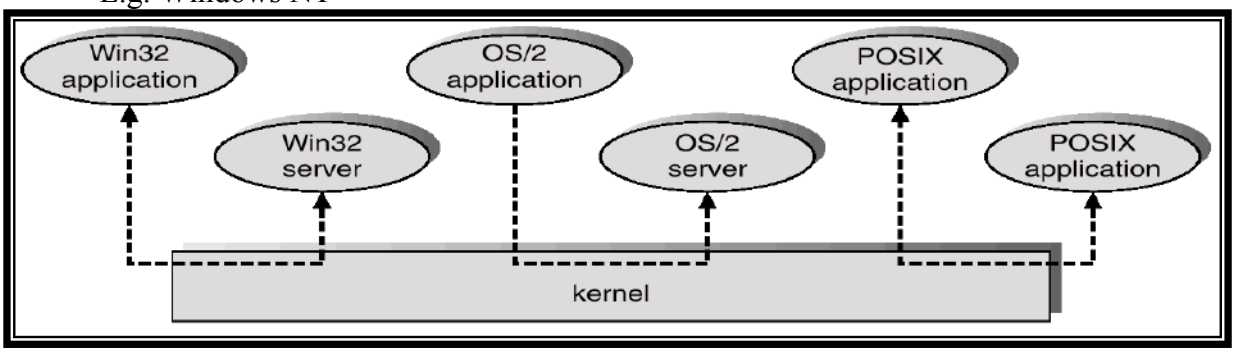


Figure 23: Client Server structure

1.35. Virtual Machines

- A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface *identical* to the underlying bare hardware.
- The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.

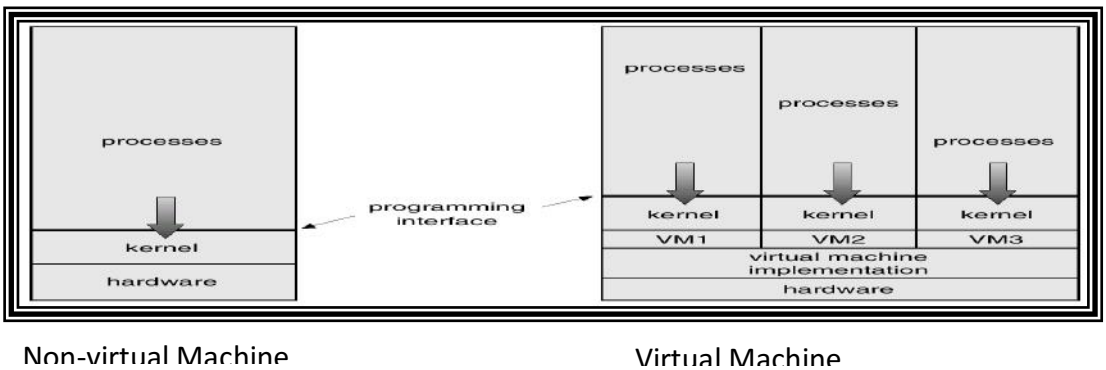


Figure 24: Virtual machines

Advantages/Disadvantages of Virtual Machines

- The virtual-machine concept provides complete protection of system resources.

  - This isolation, however, permits no direct sharing of resources.
- Perfect for OS research and development.
  - System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an *exact* duplicate to the underlying machine.

1.36. Modules Structure

- Modular Kernel created using OOP techniques
- Dynamic loading of modules
- Modern Approach
- Flexible than layered because any module can call any other module
- Efficient than micro-kernels because no need to invoke message passing.
- E.g. Solaris

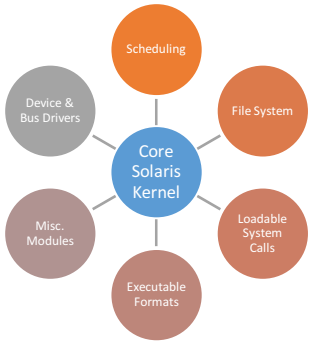


Figure 25: Modules Structure

System Design Goals

- User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast.
- System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.

1.36.1. Mechanisms and Policies

- Mechanisms determine how to do something, policies decide what will be done.
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later.

1.37. System Implementation

- Traditionally written in assembly language, operating systems can now be written in higher-level languages.
- Code written in a high-level language:
  - Can be written faster.
  - Is more compact.
  - Is easier to understand and debug.
- An operating system is far easier to *port* (move to some other hardware) if it is written in a high-level language.

1.38. System Generation (SYSGEN)

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

Summary

- An operating system is software that manages the computer hardware, as well as providing an environment for application programs to run. Perhaps the most visible aspect of an operating system is the interface to the computer system it provides to the human user.

- For a computer to do its job of executing programs, the programs must be in main memory. Main memory is the only large storage area that the processor can access directly. It is an array of words or bytes, ranging in size from millions to billions. Each word in memory has its own address. The main memory is usually a volatile storage device that loses its contents when power is turned off or lost. Most computer systems provide secondary storage as an extension of main memory.

- Secondary storage provides a form of nonvolatile storage that is capable of holding large quantities of data permanently. The most common secondary-storage device is a magnetic disk, which provides storage of both programs and data. The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

- There are several different strategies for designing a computer system. Uniprocessor systems have only a single processor, while multiprocessor systems contain two or more processors that share physical memory and peripheral devices. The most common multiprocessor design is symmetric multiprocessing (or SMP), where all processors are considered peers and run independently of one another. Clustered systems are a specialized form of multiprocessor systems and consist of multiple computer systems connected to best utilize the CPU, modern operating systems employ multiprogramming, which allows several jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute. Time-sharing systems are an extension of multiprogramming wherein CPU scheduling algorithms rapidly switch between jobs, thus providing the illusion that each job is running concurrently.

- 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/0 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.

- GNU /Linux, BSD UNIX, and Solaris are all open-source operating systems. The advantages of free software and open sourcing are likely to increase the number and quality of open-source projects, leading to an increase in the number of individuals and companies that use these projects.

- 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.

- Operating systems provide a number of services. At the lowest level, system calls allow a running program to make requests from the operating system directly. At a higher level, the command interpreter or shell provides a mechanism for a user to issue a request without writing a program. Commands may come from files during batch-mode execution or directly from a terminal when in an interactive or time-shared mode. System programs are provided to satisfy many common user requests.

- The types of requests vary according to level. The system-call level must provide the basic functions, such as process control and file and device manipulation. Higher-level requests, satisfied by the command interpreter or system programs, are translated into a sequence of system calls. System services can be classified into several categories: program control status requests, and I/0 requests. Program errors can be considered implicit requests for service.

- Once the system services are defined, the structure of the operating system can be developed. Various tables are needed to record the information that defines the state of the computer system and the status of the system's jobs.
- For a computer system to begin running, the CPU must initialize and start executing the bootstrap program in firmware. The bootstrap can execute the operating system directly if the operating system is also in the firmware, or it can complete a sequence in which it loads progressively smarter programs from firmware and disk until the operating system itself is loaded into memory and executed.

- 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.

- Spooling is useful because device access data that different rates. The buffer provides a waiting station where data can rest while the slower device catches up.

CHAPTER 2

PROCESS MANAGEMENT/THREAD MANAGEMENT

2.1. Concept of Process

A process is sequential program in execution. A process defines the fundamental unit of computation for the computer. Components of process are:

1. Object Program
2. Data
3. Resources
4. Status of the process execution.

Object program i.e. code to be executed. Data is used for executing the program. While executing the program, it may require some resources. Last component is used for verifying the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

2.1.1 Processes and Programs

- Process is a dynamic entity that is a program in execution. A process is a sequence of information executions. Process exists in a limited span of time. Two or more processes could be executing the same program, each using their own data and resources.
- Program is a static entity made up of program statement. Program contains the instructions. A program exists at single place in space and continues to exist. A program does not perform the action by itself.

Program	Process
• Consists of set of instructions in programming language	• It is a sequence of instruction execution
• It is a static object existing in a file form	• It is a dynamic object (i.e. program in execution)
• Program is loaded into secondary storage device	• Process is loaded into main memory
• The time span is unlimited	• Time span is limited
• It is a passive entity	• It is an active entity

Table 2: Difference between Program and Process

Table: Difference between Program and Process

2.2 Process State

When process executes, it changes state. Process state is defined as the current activity of the process. Fig. Below shows the general form of the process state transition diagram. Process state contains five states. Each process is in one of the states. The states are listed below.

1. ***New***: A process that just been created
2. ***Ready***: Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.

3. ***Running***: The process that is currently being executed. A running process possesses all the resources needed for its execution, including the processor.

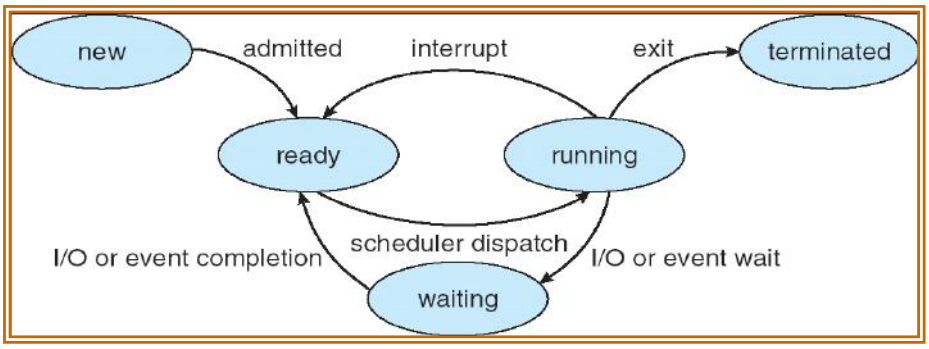


Figure 26: Diagram for Process State

4. ***Waiting***: A process that cannot execute until some event occurs such as the completion of an I/O operation. The running process may become suspended by invoking an I/O module.
5. ***Terminated(exit)***: A process that has been released from the pool of executable processes by the operating system

Whenever processes changes state, the operating system reacts by placing the process PCB in the list that corresponds to its new state. Only one process can be running on any processor at any instant and many processes may be ready and waiting state.

2.2.1 Suspended Processes

Characteristics of suspend process

1. Suspended process is not immediately available for execution.
2. The process may or may not be waiting on an event.
3. For preventing the execution, process is suspend by OS, parent process, process itself and an agent.
4. Process may not be removed from the suspended state until the agent orders the removal.

Swapping is used to move all of a process from main memory to disk. When all the process by putting it in the suspended state and transferring it to disk.

Reasons for process suspension

1. ***Swapping:*** OS needs to release required main memory to bring in a process that is ready to execute.
2. ***Timing:*** Process may be suspended while waiting for the next time interval.
3. ***Interactive user request***: Process may be suspended for debugging purpose by user.
4. ***Parent process request:*** To modify the suspended process or to coordinate the activity of various descendants.

2.3 Process Control Block (PCB)

- Each process contains the process control block (PCB). PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process. Fig. 3.2 shows the process control block.

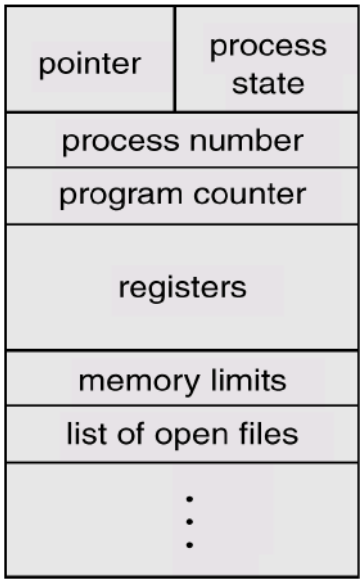


Figure 27: Process Control Block

1. ***Pointer:*** Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
2. ***Process State:*** Process state may be new, ready, running, waiting and so on.
3. ***Program Counter***: It indicates the address of the next instruction to be executed for this process.
4. ***Event information***: For a process in the blocked state this field contains information concerning the event for which the process is waiting.
5. ***CPU register***: It indicates general purpose register, stack pointers, index registers and accumulator's etc. number of register and type of register totally depends upon the computer architecture.
6. ***Memory Management Information***: This information may include the value of base and limit register. This information is useful for deallocating the memory when the process terminates.
7. ***Accounting Information***: This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.

Process control block also includes the information about CPU scheduling, I/O resource management, file management information, priority and so on. The PCB simply serves as the repository for any information that may vary from process to process.

When a process is created, hardware registers and flags are set to the values provided by the loader or linker. Whenever that process is suspended, the contents of the processor register are usually saved on the stack and the pointer to the related stack frame is stored in the PCB. In this way, the hardware state can be restored when the process is scheduled to run again.

The **process control block (PCB)** in the Linux operating system is represented by the C structure *task_struct.*

```
pid_t pid; /*  process identifier */
long state; /*  state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list__head children; /* this process's children */
struct files_struct *files;  /*  list of open files */
struct mm_struct *mm; /* address space of this process */
```

- The state of a process is represented by the field *long state.*
- Within the Linux kernel, all active processes are represented using a doubly linked list of ***task_struc***t, and the kernel maintains a pointer ***–current -*** to the process currently executing on the system.

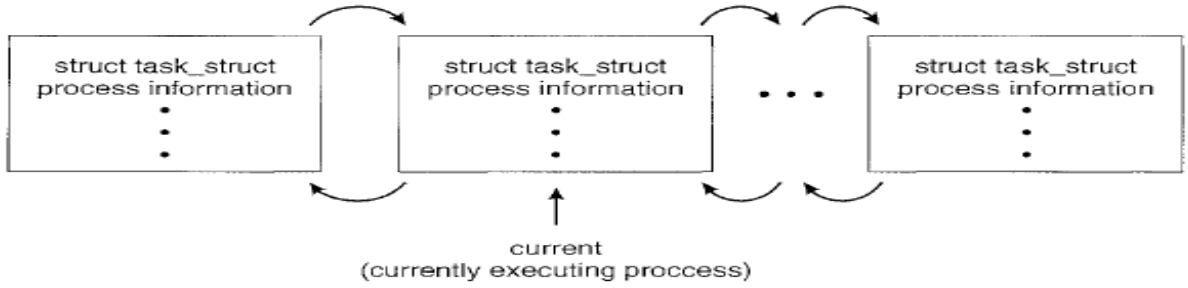


Figure 28: Active process in Linux

2.4 Process Management / Process Scheduling

Multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded process to share the CPU using time multiplexing.

- The scheduling mechanism is the part of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of particular strategy.

2.4.1. Scheduling

- When in time sharing system, CPU switches between processes
- The switch between the processes maximize the CPU utilization
- One process can get CPU time when another is waiting for I/O operations to complete
- Increases throughput
- Process scheduler selects available process for execution in CPU

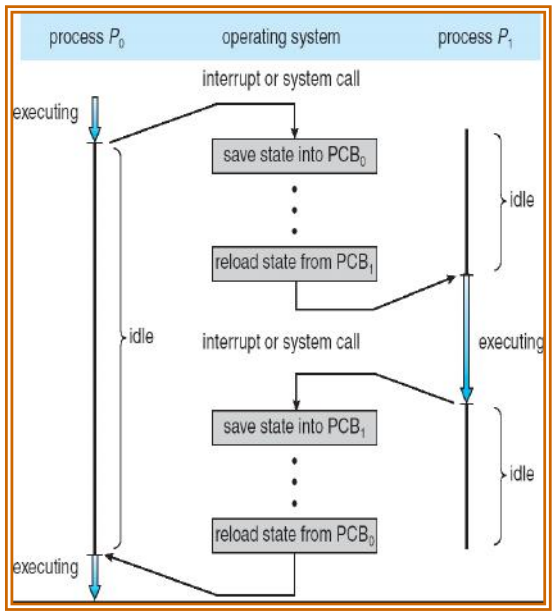


Figure 29: CPU switching between processes

2.4.2 Scheduling Queues

- When the process enters into the system, they are put into a job queue. This queue consists of all processes in the system. The operating system also has other queues.
- Device queue is a queue for which a list of processes waiting for a particular I/O device. Each device has its own device queue. Fig. 3.3 shows the queuing diagram of process scheduling. In the fig 3.3, queue is represented by rectangular box. The circles represent the resources that serve the queues. The arrows indicate the flow of processes in the system.
- ***Job queue*** – set of all processes in the system
- ***Ready queue*** – set of all processes residing in main memory, ready and waiting to execute

- ***Device queues*** – set of processes waiting for an I/O device
- Processes migrate among the various queues

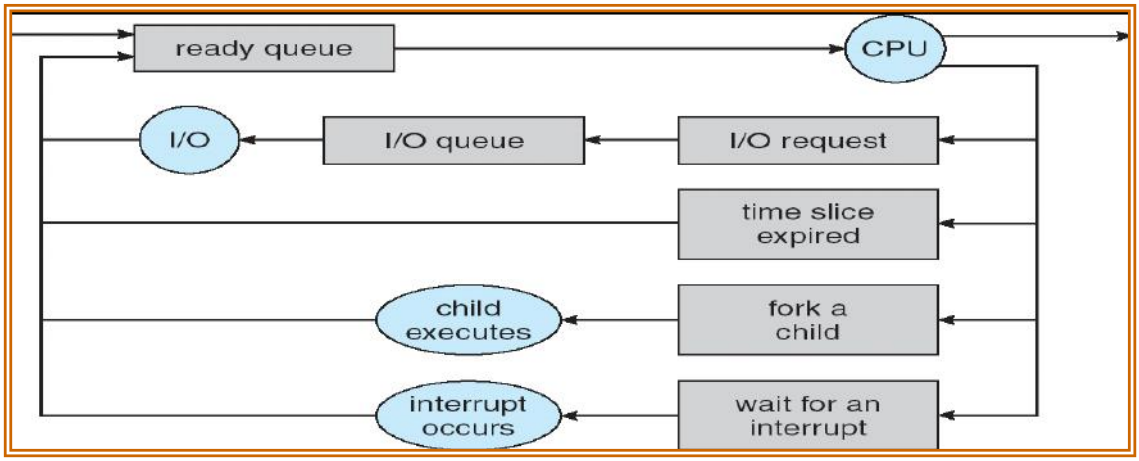
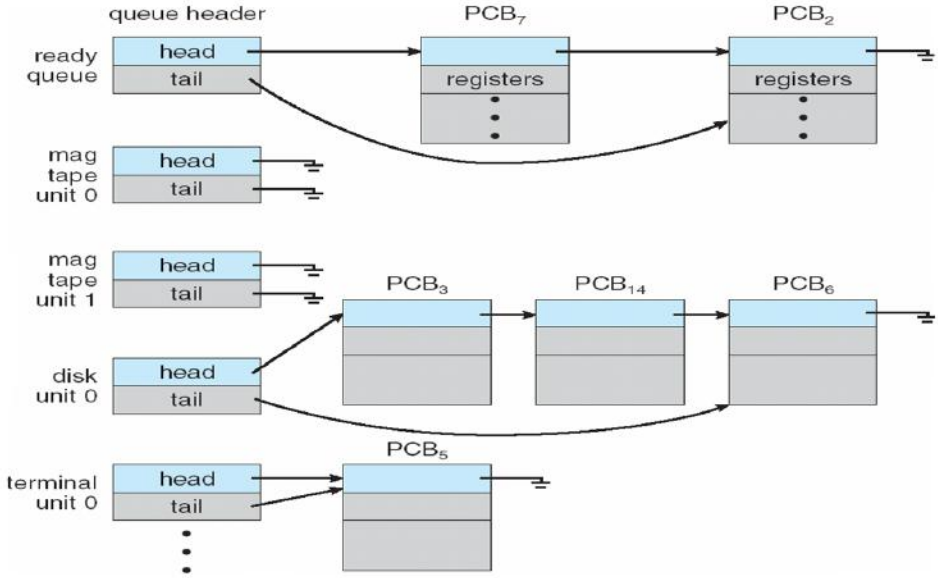


Figure 30: Queuing Diagram

Queues are of two types: ***ready queue*** and set of ***device queues***. A newly arrived process is put in the ready queue. Processes are waiting in ready queue for allocating the CPU. Once the CPU is assigned to the process, then process will execute. While executing the process, one of the several events could occur.

1. The process could issue an I/O request and then place in an I/Queue.
2. The process could create new sub process and waits for its termination.
3. The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

Ready Queue and Various I/O Device Queues



2.4.2.1 Two State Process Model

Process may be in one of two states:

a) Running
b) Not Running

When new process is created by OS, that process enters into the system in the running state.

Processes that are not running are kept in queue, waiting their turn to execute. Each entry in the queue is a printer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then select a process from the

Queue to execute.

2.4.3 Schedules

Schedulers are of three types.

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

2.4.3.1 Long Term Scheduler

It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduler. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On same systems, the long term scheduler may be absent or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is a long term scheduler.

2.4.3.2 Short Term Scheduler

It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them.

Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long tern scheduler.

2.4.3.3 Medium Term Scheduler

Medium term scheduling is part of the swapping function. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in charge of handling the swapped out-processes. Running process may become suspended by making an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process. Suspended process is move to the secondary storage is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

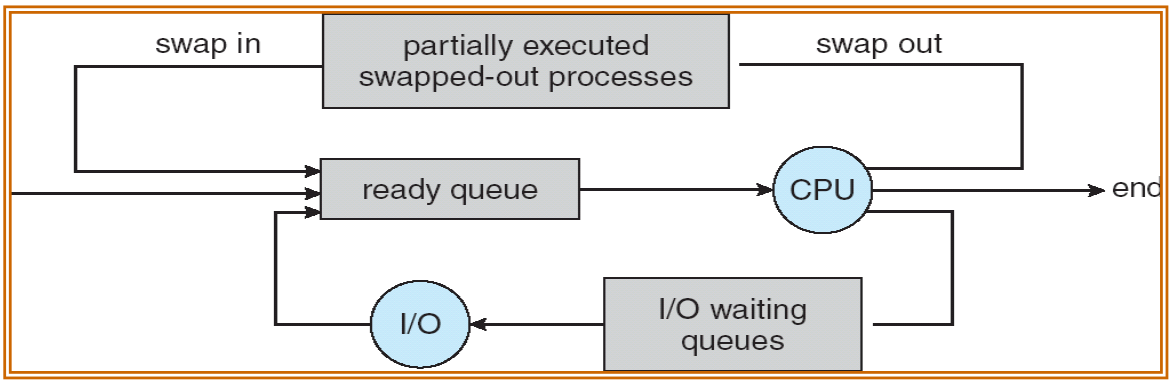


Figure 31: Queueing diagram with medium term scheduler

2.4.3.4 Comparison between Scheduler

S.N.	Long Term	Short Term	Medium Term
1	It is job scheduler	It is CPU Scheduler	It is swapping
2	Speed is less than short term scheduler	Speed is very fast	Speed is in between both
3	It controls degree of multiprogramming	Less control over degree of multiprogramming	Reduce the degree of multiprogramming.
4	Absent or minimal in time sharing system.	Minimal in time sharing system.	Time sharing system use medium term scheduler.
5	It select processes from pool and load them into memory for execution.	It select from among the processes that are ready to execute.	Process can be reintroduced into memory and its execution can be continued.
6	Process state is (New to Ready)	Process state is (Ready to Running)	
7	Select a good process, mix of I/O bound and CPU bound.	Select a new process for a CPU quite frequently.	

Table 3: Comparison of Short Term, Medium Term and Long Term Scheduler

2.5 Context Switch

When the scheduler switches the CPU from executing one process to executing another, the context switcher saves the content of all processor registers for the process being removed from the CPU in its process being removed from the CPU in its process descriptor. The context of a process is represented in the process control block of a process. Context

switch time is pure overhead. Context switching can significantly affect performance, since modern computers have a lot of general and status registers to be saved.

Content switch times are highly dependent on hardware support. Context switch requires (n + m) b*K time units to save the state of the processor with n general registers, assuming b store operations are required to save register and each store instruction requires K time units. Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time.

When the process is switched the information stored is:

1. Program Counter
2. Scheduling Information
3. Base and limit register value
4. Currently used register
5. Changed State
6. I/O State
7. Accounting

2.6 Operation on Processes

Several operations are possible on the process. Process must be created and deleted dynamically. Operating system must provide the environment for the process operation. We discuss the two main operations on processes.

1. Create a process
2. Terminate a process

2.6.1. Create Process

Operating system creates a new process with the specified or default attributes and identifier. A process may create several new sub processes. Syntax for creating new process is:

CREATE (processed, attributes)

Two names are used in the process they are parent process and child process. Parent process is a creating process. Child process is created by the parent process. Child process may create another subprocess. So it forms a tree of processes. When operating system issues a CREATE system call, it obtains a new process control block from the pool of free memory, fills the fields with provided and default parameters, and insert the PCB into the ready list. Thus it makes the specified process eligible to run the process.

When a process is created, it requires some parameters. These are priority, level of privilege, requirement of memory, access right, memory protection information etc. Process will need certain resources, such as CPU time, memory, files and I/O devices to complete the operation. When process creates a subprocess, that subprocess may obtain its resources directly from the operating system. Otherwise it uses the resources of parent process.

When a process creates a new process, two possibilities exist in terms of execution.

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

For address space, two possibilities occur:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

UNIX examples

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

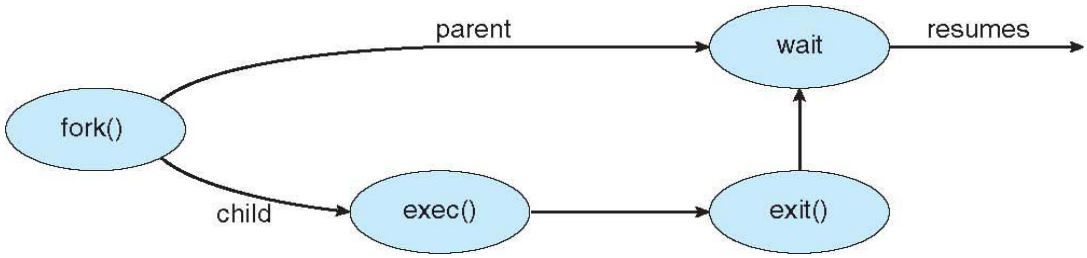


Figure 32: Fork () Process

2.6.2 Terminate a Process

- Process executes last statement and then asks the operating system to delete it using the **exit ()** system call.
  - Returns status data from child to parent (via **wait()**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort ()** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
  - The task given to the child is no longer required.
  - Child has exceeded its usage of some of the resources that it has been allocated.
  - Operating system does not allow a child to continue if its parent terminates.
  - Some operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
    - **Cascading termination.** All children, grandchildren, etc. are terminated.
    - The termination is initiated by the operating system.
  - The parent process may wait for termination of a child process by using the **wait ()** system call**.** The call returns status information and the pid of the terminated process

    pid = wait (&status);
  - If no parent waiting (did not invoke **wait()**) process is a **zombie**
  - If parent terminated without invoking **wait** , process is an **orphan**

- Conditions due to which process terminates:
  - ***Normal Exit (Voluntary)***
    - After completion of the task
  - ***Error Exit (Voluntary)***
    - Due to unavailable program compilation
    - Asks user to try again with correct parameter or exit
  - ***Fatal Error (Involuntary)***
    - Due to program bug. Eg: illegal instruction, non-existence memory..
  - ***Killed by another process (Involuntary)***
    - Process executes system call to kill another process
    - Killing process must have authorization to do so

2.7 Co-operating Processes

Co-operating process is a process that can affect or be affected by the other processes while executing. If suppose any process is sharing data with other processes, then it is called co-operating process. Benefit of the co-operating processes are:

1. Sharing of information
2. Increases computation speed
3. Modularity
4. Convenience

1. ***Co-operating processes*** share the information: Such as a file, memory etc. System must provide an environment to allow concurrent access to these types of resources.
   - Computation speed will increase if the computer has multiple processing elements are connected together. System is constructed in a modular fashion. System function is divided into number of modules.
   - ***Cooperating*** **process** can affect or be affected by the execution of another process
   - Share data with another process/es
   - Advantages of process cooperation
   - Information sharing (eg: file sharing between processes)
   - Computation speed-up (subtasks running parallel with multiple CPUs or I/O channels)
   - Modularity (Threads/Modules)
   - Convenience (multitasks on same time)
2. ***Independent*** **process** cannot affect or be affected by the execution of another process.
   - Do not share data with any other process

2.7.1. Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- ***Message system*** – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
- **send**(*message*) – message size fixed or variable
- **receive**(*message*)
  - If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them

- exchange messages via send/receive

• Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

2.7.1.1. Communications Models

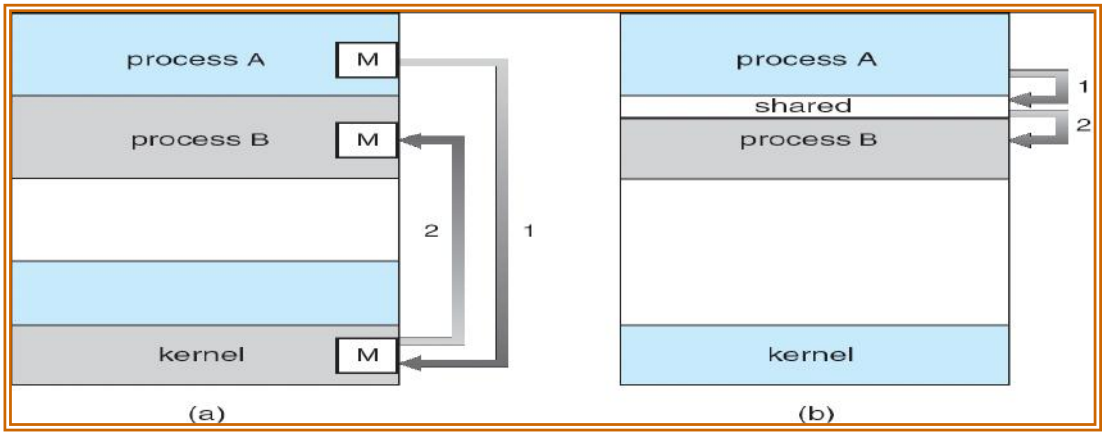


Figure 33: Communication Model (a) Message Passing b) Shared Memory

Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users' processes not the operating system.
- A major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(*message*)
  - receive(*message*)
- The *message* size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a *communication link* between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send(*A, message*) – send a message to mailbox A
  - receive(*A, message*) – receive a message from mailbox A
- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
  - Solutions
    - Allow a link to be associated with at most two processes
    - Allow only one process at a time to execute a receive operation
    - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was

Synchronization

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

Buffering

Queue of messages attached to the link; implemented in one of three ways

1. Zero capacity – 0 messages
    - Sender must wait for receiver (rendezvous)
2. Bounded capacity – finite length of *n* messages
    - Sender must wait if link full
3. Unbounded capacity – infinite length
    - Sender never waits

2.8 THREAD MANAGEMENT

2.8.1. Introduction of Thread

A thread is a flow of execution through the process code, with its own program counter, system registers and stack. Threads are a popular way to improve application performance through parallelism. A thread is sometimes called a light weight process.

Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process. Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.

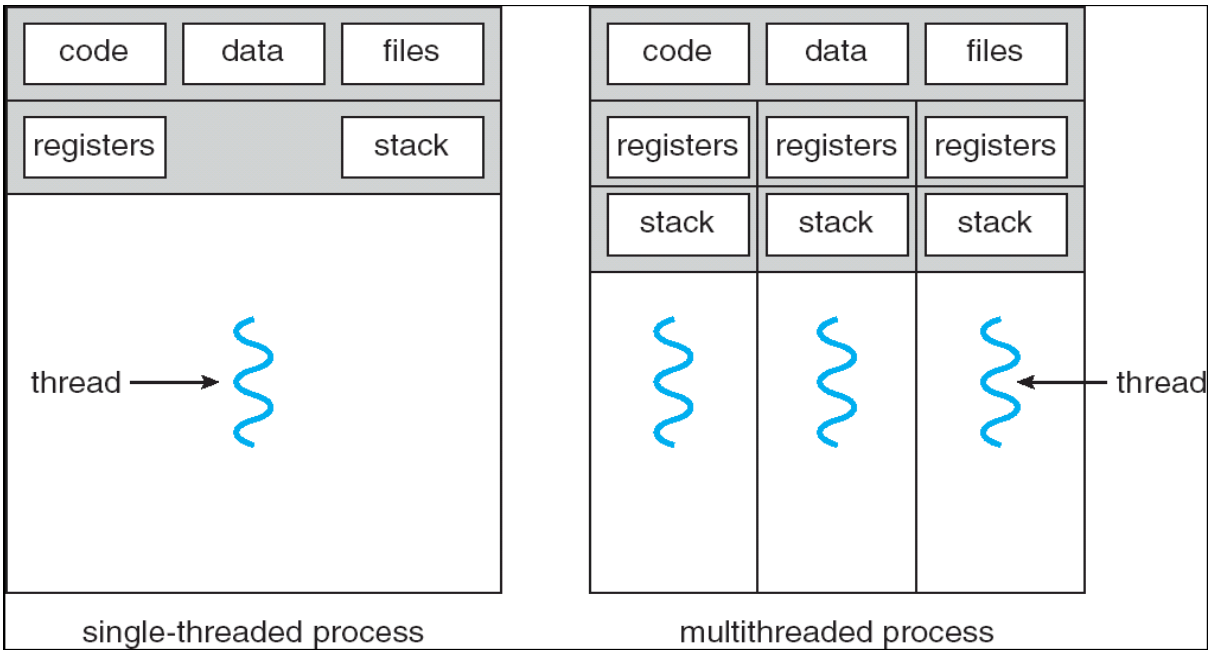


Figure 34: Single and multithreaded process.

Threads have been successfully used in implementing network servers. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

2.8.2. Multithreaded Server Architecture

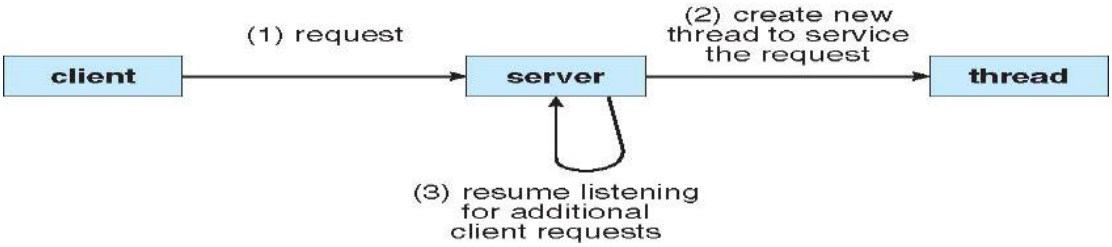


Figure 35: Multithreaded Server Architecture

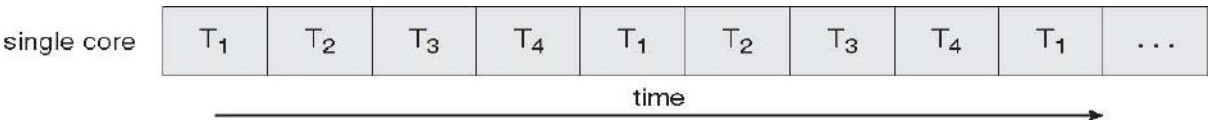


Fig: Concurrent Execution on a Single Core system

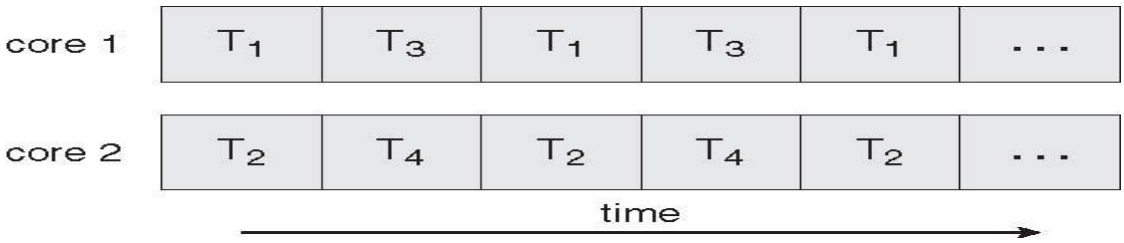


Figure 36: Parallel Execution on a Multi Core system

2.8.3. Why Threads? (Benefits)

- Process with multiple threads makes a great server (e.g. print server)
- Increase responsiveness, i.e. with multiple threads in a process, if one threads blocks then other can still continue executing
- Sharing of common data reduce requirement of inter-process communication
- Proper utilization of multiprocessor by increasing concurrency
- Threads are cheap to create and use very little resources
- Context switching is fast (only have to save/reload PC, Stack, SP, Registers)

2.8.4. Process vs Thread

Process	Thread
• Doesn't share memory (loosely coupled)	• Shares memories and files (tightly coupled)
• Creation is time consuming	• Fast
• Process is called heavy weight process.	• Thread is called light weight process.
• Process switching needs interface with operating system.	• Thread switching does not need to call an operating system and cause an interrupt to the Kernel.
• In multiple process implementation each process executes the same code but has its own memory and file resources.	• All threads can share same set of open files, child processes.
• If one server process is blocked no other server process can execute until the first process unblocked.	• While one server thread is blocked and waiting, second thread in the same task could run.
• Multiple redundant process uses more resources than multiple threaded.	• Multiple threaded process uses fewer resources than multiple redundant process.
• In multiple process each process operates independently of the others.	• One thread can read, write or even completely wipe out another threads stack.

Table 4 : Process vs Thread

2.8.5 Types of Thread

Threads is implemented in two ways:

1. User Level
2. Kernel Level

2.8.5.1 User Level Thread

In a user thread, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.

User level threads are generally fast to create and manage.

Advantage of user level thread over Kernel level thread:

1. Thread switching does not require Kernel mode privileges.
2. User level thread can run on any operating system.
3. Scheduling can be application specific.
4. User level threads are fast to create and manage.

Disadvantages of user level thread:

1. In a typical operating system, most system calls are blocking.
2. Multithreaded application cannot take advantage of multiprocessing

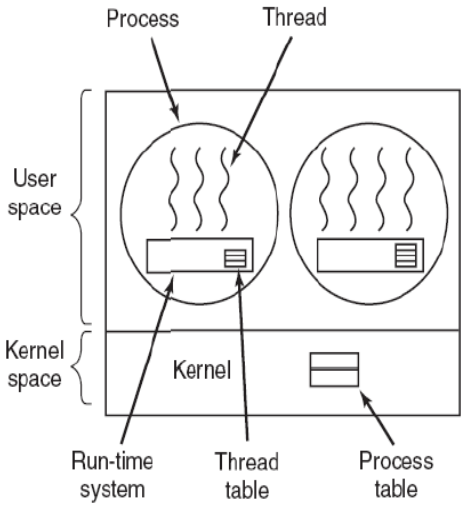


Figure 37: User Level Thread

2.8.5.2. Kernel Level Threads

In Kernel level thread, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process. The Kernel maintains context information for the process as a whole and for individual's threads within the process.

Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

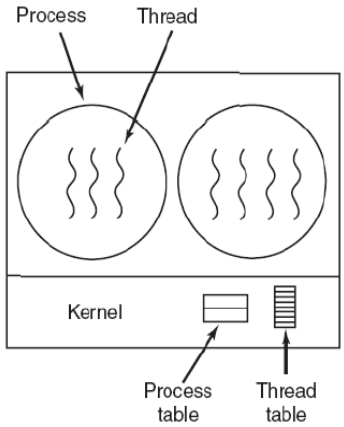


Figure 38: Kernel Level thread

Advantages of Kernel level thread:

1. Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
2. If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
3. Kernel routines themselves can multithreaded.

Disadvantages:

1. Kernel threads are generally slower to create and manage than the user threads.
2. Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

2.8.5.4.Combined or Hybrid Threads

- Combines both user level and kernel level threads
- Thread creation is done in user space

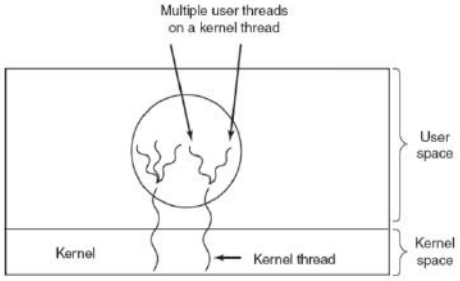


Figure 39: Hybrid Thread

2.8.5.5 Advantages of Thread

1. Thread minimizes context switching time.
2. Use of threads provides concurrency within a process.
3. Efficient communication.
4. Economy- It is more economical to create and context switch threads.
5. Utilization of multiprocessor architectures –The benefits of multithreading can be greatly increased in a multiprocessor architecture.

2.8.6 Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.

Multithreading models are three types:

1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

2.8.6.1 Many to Many Model

In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

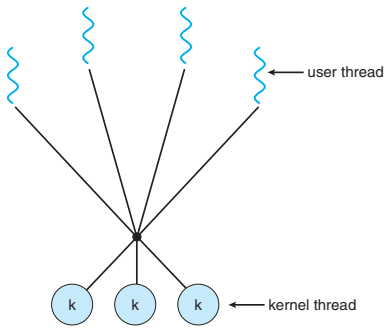


Figure 40: Many-to-Many Model

Fig. shows the many to many models. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.

2.8.6.2. Many to One Model

Many to one model maps many user level threads to one Kernel level thread.

Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user level thread libraries are implemented in the operating system, that system does not support Kernel threads use the many to one relationship modes.

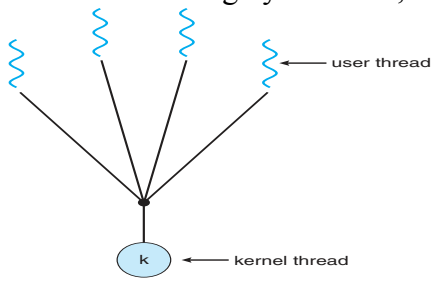


Figure 41: Many to one model.

2.8.6.3. One to One Model

There is one to one relationship of user level thread to the kernel level thread. Fig. 4.5 shows one to one relationship model. This model provides more concurrency than the many to one model.

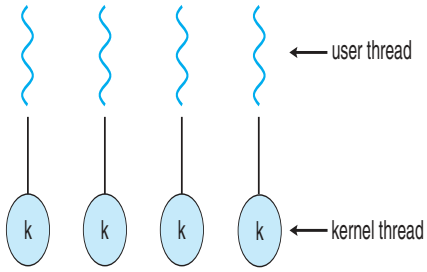


Figure 42: One to One Thread

It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and windows 2000 use one to one relationship model.

2.8.7 Difference between User Level & Kernel Level Thread

Sn.	User Level Threads	Kernel Level Thread
1	• User level thread are faster to create and manage.	• Kernel level thread are slower to create and manage.
2	• Implemented by a thread library at the user level.	• Operating system support directly to Kernel threads.
3	• User level thread can run on any operating system.	• Kernel level threads are specific to the operating system.
4	• Support provided at the user level called user level thread.	• Support may be provided by kernel is called Kernel level threads.
5	• Multithread application cannot take advantage of multiprocessing.	• Kernel routines themselves can be multithreaded.

Table 5: Difference Between User Level Threads and Kernel Level threads

2.8.8. Threading Issues

In a multithreaded program environment, fork and exec system calls is changed. UNIX system have two version of fork system calls. One call duplicates all threads and another that duplicates only the thread that invoke the fork system call. Whether to use one or two version of fork system call totally depends upon the application. Duplicating all threads is unnecessary, if exec is called immediately after fork system call.

Thread cancellation is a process of thread terminates before its completion of task. For example, in multiple thread environment, thread concurrently searching through a database. If anyone thread returns the result, the remaining thread might be cancelled.

Thread Cancellation is of two types

1. Asynchronous cancellation
2. Synchronous cancellation

In asynchronous cancellation, one thread immediately terminates the target thread. Deferred cancellation periodically check for terminate by target thread. It also allow the target thread to terminate itself in an orderly fashion. Some resources are allocated to the thread. If we cancel the thread, which update the data with other thread. This problem may face by asynchronous cancellation system wide resource are not free if threads cancelled asynchronously. Most of the operating system allow a process or thread to be cancelled asynchronously.

2.8.8.1 Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
    - A **signal handler** is used to process signals
    - Signal is generated by particular event
    - Signal is delivered to a process
    - Signal is handled
- **Synchronous vs. asynchronous signal**
    - Synchronous Signal include illegal memory access and division by 0.
    - When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control> <C>) and having a timer expire.
- Default vs. user-defined handlers
- Default handler run by the kernel.
- User-defined: Handled differently(ignored or terminating the program)
- Options to deliver signals in multithreaded programs:
    - Deliver the signal to the thread to which the signal applies
    - Deliver the signal to every thread in the process
    - Deliver the signal to certain threads in the process
    - Assign a specific thread to receive all signals for the process
    - ex:

- kill(aid_t aid, int signal)
- pthread_kill(pthread_t tid, int signal)

2.8.8.2. Thread Pools

- Create a number of threads in a pool where they await work
- **Advantages:**
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Ex:
  - QueueUserWorkItem(LPTHREAD_START_ROUTINE Function, PVOID Param, ULONG Flags) in Win32 API
  - java.util.concurrent package in Java 1.5

2.8.8.3 Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- An intermediate data structure called *LWP* (Lightweight Process) between user thread and kernel thread
- This communication allows an application to maintain the correct number kernel threads

2.9.Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

CPU Bound (or compute bound): Some process spend most of their time computing. These processes tend to have long CPU burst and thus infrequent I/O waits. Example: Matrix multiplication

I/O Bound: Some process spend most of their time waiting for I/O. They have short CPU bursts and thus frequent I/O waits. Example: Firefox

CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.

- Both start and end by CPU burst.
- Switches between CPU burst and I/O burst multiple times before termination

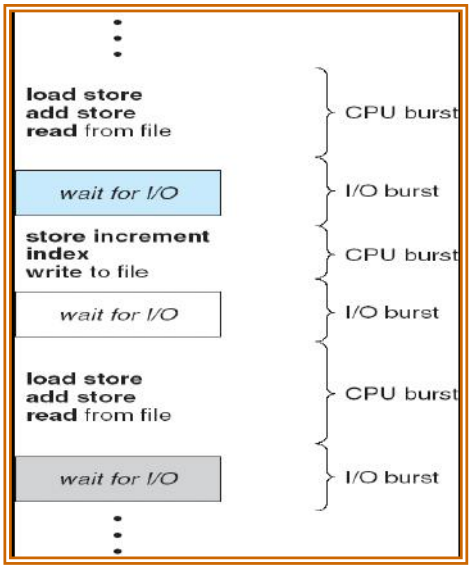


Figure 43: Alternating Sequence of CPU and I/O bursts

The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Above fig).

2.9.1. 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.

2.9.2. Preemptive and non-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

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. Unfortunately, preemptive scheduling incurs a cost associated with access to shared data.
Consider the case of two processes that share data. While one is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state. In such situations, we need new mechanisms to coordinate access to shared data.
Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues.

Non-Preemptive Scheduling	Preemptive Scheduling
• Once the CPU is allocated to a process, it cannot be taken away from that process until termination	• CPU can be taken away before the completion
• No preference for higher priority job	• CPU switches to higher preference job from lower preference job from middle
• Fair treatment to all the processes	• Not fair, CPU either switches because of time constraints or due to higher priority process
• Cheap to implement	• Costlier to implement
• E.g.: FCFS algorithm	• E.g.: Round Robin (RR) algorithm

Table 6: Preemptive vs Non-preemptive scheduling

2.9.3. Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- ***Dispatch latency*** – time it takes for the dispatcher to stop one process and start another running

2.9.4. Scheduling Criteria

Many criteria have been suggested for comparison of CPU scheduling algorithms.

CPU utilization: Average time during which CPU is busy executing user programs and/or system programs. The more the better. We have to keep the CPU as busy as possible. It may range from 0 to 100%. In a real system it should range from 40 – 90 % for lightly and heavily loaded system.

Throughput: It is the measure of work in terms of number of process completed per unit time. For long process this rate may be 1 process per hour, for short transaction, throughput may be 10 process per second.The higher the better.

Turnaround Time: It is the sum of time periods spent in waiting to get into memory, waiting in ready queue, execution on the CPU and doing I/O. The interval form the time of submission of a process to the time of completion is the turnaround time. Waiting time plus the service time.

Turnaround time (TAT) = Time of completion of job - Time of submission of job. (Waiting time + service time or burst time)

Waiting time: Amount of time a process has been waiting in the ready queue

- Hence, WT = TAT – Processing Time
- Minimum is better.

Response time: in interactive system the turnaround time is not the best criteria. Response time is the amount of time it takes to start responding, not the time taken to output that response. Minimum is better.

2.9.5. Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

2.9.6. Scheduling Algorithms Goals

- All Systems:
  - Fairness:
    - Fair share of CPU to every process
  - Policy Enforcement:
    - Stated policy is implemented properly
  - Balance:
    - Keeping all resources equally busy
- Batch System:
  - Throughput:
    - Maximize jobs per unit time
  - Turnaround Time:
    - Minimization of time between submission and termination
  - CPU Utilization:
    - Keep CPU busy all the time
- Interactive System:
  - Response Time:
    - Quick response to request
  - Proportionality:
    - Meet user's expectation

- Real Time System:
  - Meeting Deadlines:
  - Avoiding losing data
- Predictability:
- Avoid quality degradation

2.9.7. Job (Process) Scheduling Algorithms

1. First-Come-First-Serve (FCFS)
2. Shortest Job First (SJF)
3. Shortest Remaining Time Next (STRF)
4. Round Robin (RR)
5. Priority (preemptive and non-preemptive)
6. Multilevel Queue

2.9.7.1. First Come First Server

FCFS is the simplest **non-preemptive** algorithm. Processes are assigned the CPU in the order they request it. That is the process that requests the CPU first is allocated the CPU first. The implementation of FCFS is policy is managed with a FIFO (First in first out) queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. On the negative side, the average waiting time under the FCFS policy is often quite long.

Advantages:

1. Easy to understand and program. With this algorithm a single linked list keeps track of all ready processes.
2. Equally fair.
3. Suitable especially for Batch Operating system.

Disadvantages:

1. FCFS is not suitable for time-sharing systems where it is important that each user should get the CPU for an equal amount of arrival time.

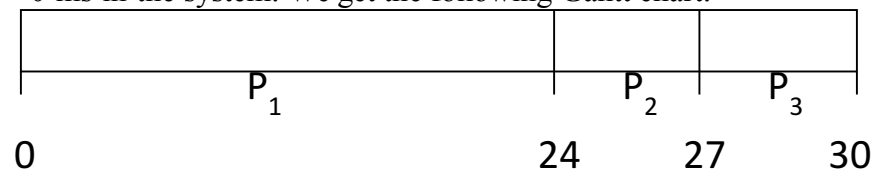
Consider the following set of processes having their burst time mentioned in milliseconds. CPU burst time indicates that for how much time, the process needs the cpu.

Process	Burst Time
P1	24
P2	3
P3	3

Calculate the average waiting time and Average turnaround time if the processes arrive in the order of:

a) P1, P2 , P3 and
b) P2, P3, P1

a) The processes arrive the order P1, P2, P3. Let us assume they arrive in the same time at 0 ms in the system. We get the following Gantt chart.



Waiting time for P1= 0ms, for P2 = 24 ms, for P3 = 27ms
Average waiting time (AWT): (0+24+27)/3= 17ms

Turnaround time (TAT) = waiting time (WT) + Burst Time (BT)
P1=0+24 = 24
P2= 24+3=27
P3= 27+3= 30
Average Turnaround Time (ATAT) = (24+27+30)/3 = 81/3 = 27ms

b) If the process arrive in the order P2, P3, P1
we get the following Gantt chart

Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
Average waiting time (AWT): (6 + 0 + 3)/3 = 3ms

Turnaround time (TAT) = waiting time (WT) + Burst Time (BT)
P1=6+24 = 30
P2= 0+3=3
P3= 3+3= 6
Average Turnaround Time (ATAT) = (30+3+6)/3 = 39/3 = 13ms

In above example:
Short processes coming later than long process have to wait more for CPU. This effect is called ***Convoy effect.***

2.9.7.2. Shortest Job First (SJF)

- Allocate the CPU to the process with least CPU burst time
- If two processes have same CPU burst, then FCFS is used to break the tie
- Two schemes;
- Non Preemptive :
  - Once CPU given to the process it cannot be preempted until completes its CPU burst
- Preemptive (also called Shortest Remaining First -SRTF):
  - If a new process arrives with CPU burst length less than remaining time of current executing process, preempt.

- Preemptive SJF scheduling is sometimes called **Shortest Remaining Time First** scheduling. With this scheduling algorithms the scheduler always chooses the process whose remaining run time is shortest.

- *When a new job arrives its total time is compared to the current process remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job is started. This scheme allows new short jobs to get good service.*

Process	Arrival Time
P_1	0.0
P_2	2.0
P_3	4.0
P_4	5.0

- SJF (non-preemptive)
- Average waiting time = (0 + 6 + 3 + 7)/4 = 4ms

Process	Arrival Time
P_1	0.0
P_2	2.0
P_3	4.0
P_4	5.0

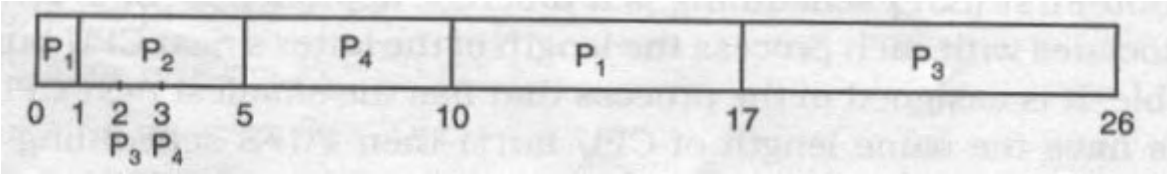
- SJF (preemptive)
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Q. Calculate the average waiting time in l). Preemptive SJF and 2). Non Preemptive SJF

Note: SJF Default: (Non Preemptive)

Process	Arrival Time	Burst Time
Pl	**0**	**8**
P2	**l**	**4**
P3	**2**	**9**
P4	**3**	**5**

Preemptive SJF (Shortest Remaining Time First)



At t=0ms only one process Pl is in the system, whose burst time is 8ms; starts its execution. After lms i.e., at t=l, new process P2 (Burst time= 4ms) arrives in the ready queue. Since its burst time is less than the remaining burst time of Pl (7ms) Pl is preempted and execution of P2 is started.

Again at t=2, a new process P3 arrive in the ready queue but its burst time (9ms) is larger than remaining burst time of currently running process (P2 3ms). So P2 is not preempted and continues its execution. Again at t=3, new process P4 (burst time 5ms) arrives. Again for same reason P2 is not preempted until its execution is completed.

Waiting time of P1 = 0ms + (10 – 1) ms = 9ms
Waiting time of P2= 1ms – 1ms = 0ms
Waiting time of P3= 17ms – 2ms = 15ms
Waiting time of P4: 5ms – 3ms = 2ms
Average waiting time (AWT) = (9+0+15+2)/4 = 6.5ms

Turn Around time (TAT) = Waiting time + burst time
P1= 9+8 = 17ms
P2=0+4=4ms
P3=15+9=24ms
P4=2+5=7ms
Average Turnaround time (ATAT) = (17+4+24+7)/4 = 52/4 = 13ms

SJF Advantages:

- Optimal – gives minimum average waiting time for a given set of processes.

SJF Disadvantages:

- Difficult to know the length of next CPU burst time
- Big jobs are waiting for CPU, which may result in **aging problem.**

2.9.7.3. Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in the FCFS order.

Assigning Priority

1. To prevent high priority process from running indefinitely the scheduler may decrease the priority of the currently running process at each clock interrupt. If this causes its priority to drop below that of the next highest process, a process switch occurs.
2. Each process may be assigned a maximum time quantum that is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

Priorities can be assigned statically or dynamically. For UNIX system there is a command nice for assigning static priority. It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class.

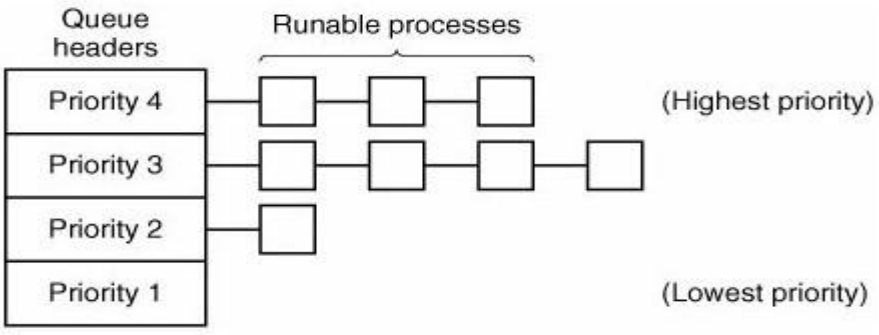


Figure 44: A scheduling algorithm with four Priority classes

- Gives preferential treatment to important jobs.
- Allows the programs with the highest priority to be processed first
- If two or more jobs with equal priority, then implements FCFS algorithm
- The priority determination is affected by:
  - Resource requirement
  - Processing time
  - Total spent time on the system
- Types:
  - Preemptive Priority Scheduling
  - Non-preemptive Priority Scheduling
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer)
  - Preemptive
  - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- **Problem**:
  - **Starvation** – low priority processes may never execute
- Solution:
  - **Aging** – as time progresses increase the priority of a waiting process

Process	Burst Time	Priority	Arrival Time
P1	10	3	0
P2	5	2	1
P3	2	1	2
Note: Priority 1 means highest priority & 3 means lowest priority			

The Gantt chart is

P1	P2	P3	P2	P1

0 1 2 4 8 17

AWT = {0+(8-1)+0+(4-2)+0}/3 = 9/3 = 3 ms

Preemptive Priority (Problems)

- Indefinite blocking of low priority processes by high priority processes called ***Starvation***
- Completion of a process within finite time is not guaranteed
- Solution:
- ***Aging:*** Increasing priority of process that has waited for long time. Older process gets high priority and hence can complete in finite time.

2.9.7.4. Non-Preemptive Priority Scheduling

Process	Arrival Time (AT)	Burst Time	Priority
P0	0	10	5
P1	1	6	4
P2	3	2	2
P3	5	4	0

AWT = {0+(10-5)+(14-3)+(16-1)}/4 = 31/4 = 7.75 ms

2.9.7.5. Round-Robin Scheduling Algorithms:

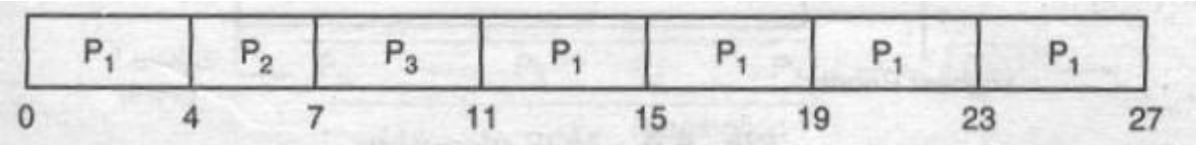
- Preemptive FCFS algorithm
- One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).
- In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum.
- If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.
- If the process has blocked or finished before the quantum has elapsed the CPU switching is done.
- Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.
- The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.
- In any event, the average waiting time under round robin scheduling is on quite long.
- Each process gets a small unit of CPU time called time quantum (10-100 milliseconds).
- Let,
    - n = number of processes,
    - q = time quantum, then
    - no process runs more than one time slice (i.e. q)
    - no process waits more than (n-1)*q time units

- If a running process releases control due to some I/O, another process is scheduled to run
- Note:
  - q Must be large w.r.t. context switch, otherwise overhead is too high.

- Consider the following set of processes that arrives at time 0 ms.

Process	Burst Time
Pl	20
P2	3
P3	4

If we use time quantum of 4ms then calculate the average waiting time using R-R scheduling.



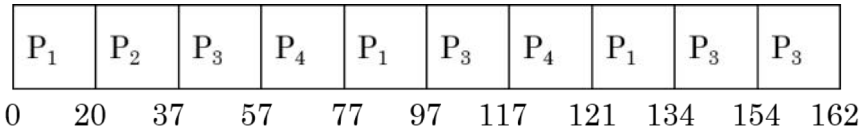
According to R-R scheduling processes are executed in FCFS order. So, firstly Pl (burst time=20ms) is executed but after 4ms it is preempted and new process P2 (Burst time = 3ms) starts its execution whose execution is completed before the time quantum. Then next process P3 (Burst time=4ms) starts its execution and finally remaining part of Pl gets executed with time quantum of 4ms.

Waiting time of Process Pl: 0ms + (11 - 4) ms = 7ms
Waiting time of Process P2: (4-0) = 4ms
Waiting time of Process P3: (7-0) =4ms
Average Waiting time: (7+4+7)/3=6 ms

Example: Time Quantum = 20 ms

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

The Gantt chart is:



Typically, higher average turnaround than SJF, but better *response*.
Calculate: AWT and ATAT vourself.

2.9.8. Time Quantum and Context Switch Time

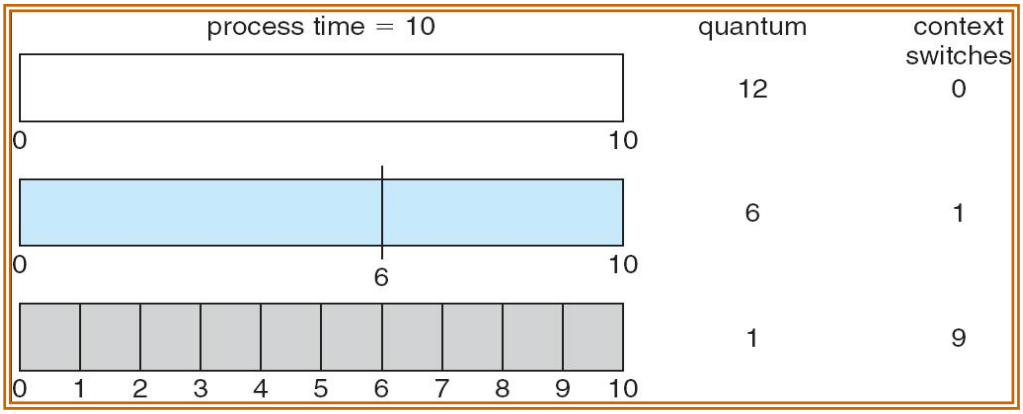


Figure 45: How a smaller time quantum increases context switches

In software, we need also to consider the effect of context switching on the performance of RR scheduling. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly**.**

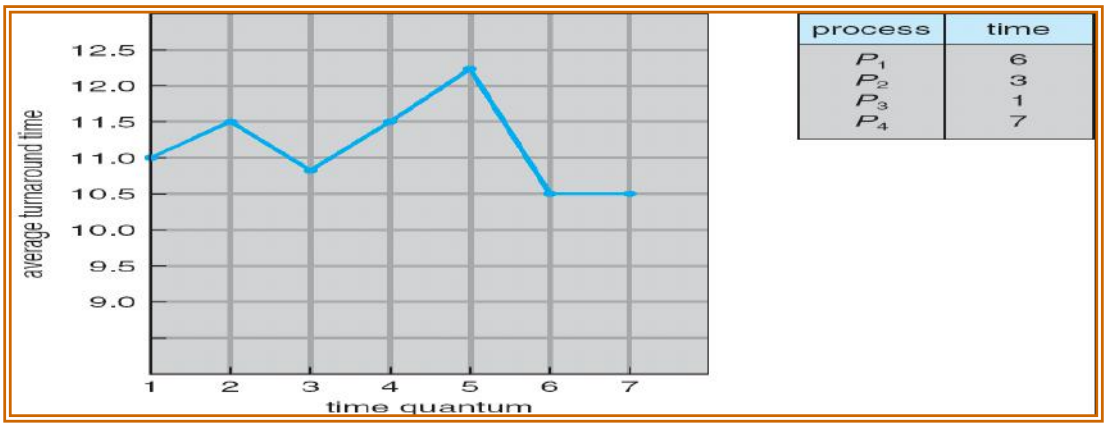


Figure 46: Turnaround time varies with quantum time

2.9.9. Turnaround Time Varies with the Time Quantum

Fig: Illustration of turnaround time varying with the time quantum

Turnaround time also depends on the size of the time quantum. As we can see from Figure, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context switch time, it should not be too large. If the time quantum is too large, RR scheduling degenerates to an FCFS policy. *A* rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

2.9.10 Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm,
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues.
- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR; 20% to background in FCFS

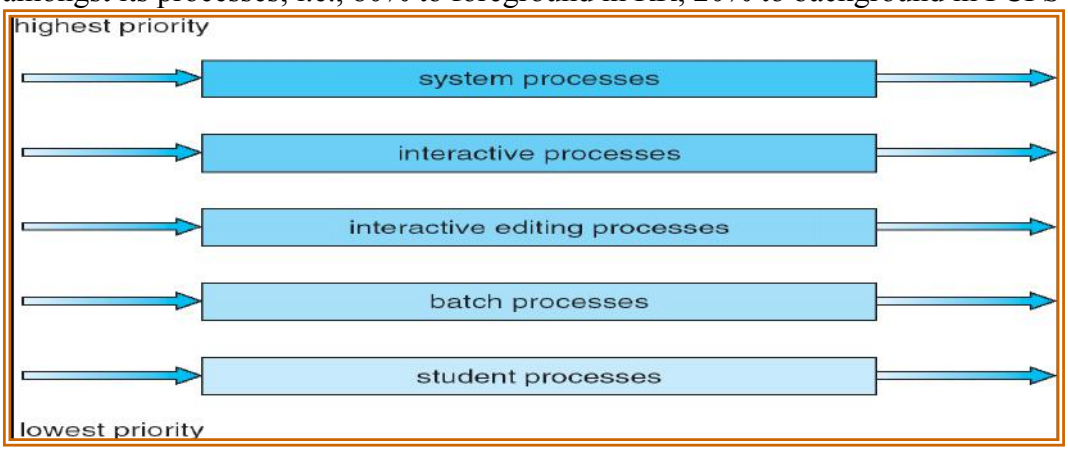


Figure 47: Multi Level Queue Scheduling

Let us look at an example of a multilevel queue scheduling algorithm with five queues, listed below in the order of priority.

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

Each queue has absolute priority over lower priority queues. No processes in the batch queue, for example could run unless the queue for System processes, interactive processes and interactive editing processes were all empty. If an interactive editing process enters the ready queue while a batch process was running the batch process would be preempted.
Another possibility is to time slice between the queues. For instance foreground queue can be given 80% of the CPU time for RR scheduling among its processes, whereas the background receives 20% of the CPU time.

2.9.11. Multilevel Feedback Queue

- A process can move between the various queues
  - aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process

- method used to determine which queue a process will enter when that process needs service.

2.9.12. Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
- A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 ms
  - If it does not finish in 8 ms, job is moved to queue Q_1
- At Q_1 job is also served FCFS and receives 16 additional ms
  - If it still does not complete, it is preempted and moved to queue Q_2

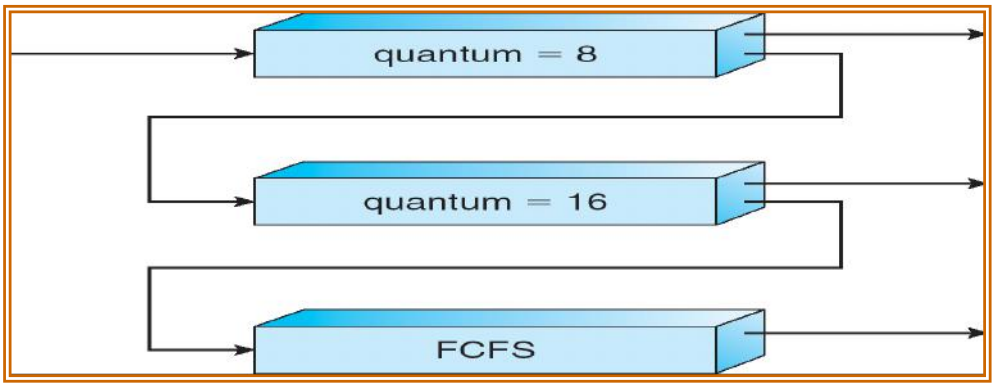
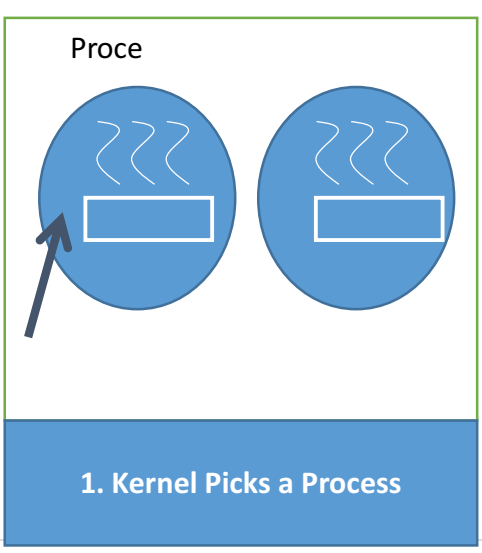
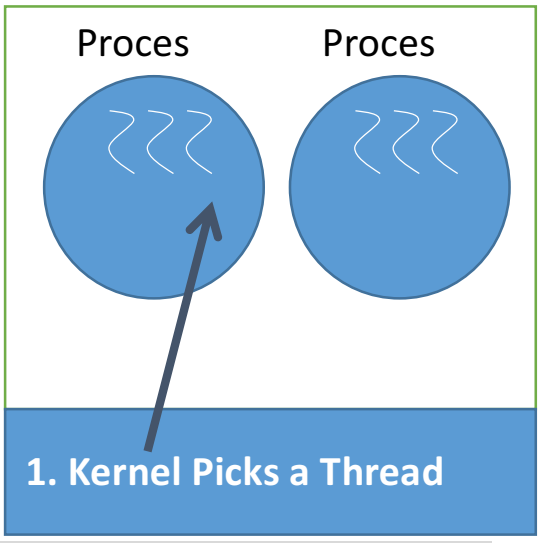


Figure 48: Example of Multilevel feedback queue

2.9.13. Thread Scheduling

- User-level threads and Kernel-level threads
- All the scheduling algorithms in thread supporting OS are Kernel-level scheduling not processes
- User-level threads are managed by a thread library, and the kernel is unaware of them
- For user level thread, kernel only picks a process and threads from the process can compete for CPU
- For Kernel level thread, Kernel picks a thread to be executed on CPU





User level Thread
Possible: A1, A2, A3, A1, A2, A3
Not-Possible: A1, B1, A2, B2, A3, B3

Kernel Level Thread
Possible: A1, A2, A3, A1, A2, A3
Also Possible: A1, B1, A2, B2, A3, B3

2.10. Process Synchronization

2.10.1 Principle of Concurrency

In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution. Even parallel processing is not achieved, and ever though there is a certain amount of overhead involved in switching back and forth between processes, interleaved execution provides major benefits in processing efficiency and in program structuring. In a multiple processor system, it is possible not only to interleave the execution of multiple processes but also to overlap them. It is assumed, it may seem that interleaving and overlapping represent fundamentally different modes of execution and present different problems. In fact, both techniques can be viewed as examples of concurrent processing, and both present the same problems. The relative speed of execution of processes It depends on activities of other processes, the way in which the operating system handles interrupts, and the scheduling policies of the operating, system.

There are quite difficulties:

1. The sharing of global resources. For example, if two processes both make use of the same global variable and both perform reads and writes on that variable, then the order in which the various reads and writes are executed is critical.
2. It is difficult for the operating system to manage the allocation of resources optimally.
3. It is very difficult to locate a programming error because results are typically not deterministic and reproducible.

2.10.2 Producer-Consumer Problem

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Consider a problem: **Producer-Consumer Problem**
- A producer process produces information that is consumed by a consumer process.
- For example: a compiler may produce assembly code, which is consumed by an assembler.
- The assembler, in turn, may produce object modules, which are consumed by the loader.
- The producer-consumer problem also provides a useful metaphor for the client-server paradigm.
- We generally think of a server as a producer and a client as a consumer.
- One solution to the producer-consumer problem uses shared memory called ***buffer***.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two processes share a common, fixed-sized buffer
- Producer puts information into the buffer
- Consumer takes information from buffer
    - ***unbounded-buffer*** places no practical limit on the size of the buffer

- Another Solution Could be:

2.10.2.1 Bounded-Buffer – Shared-Memory Solution

- The consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.
- The shared buffer is implemented as a *circular array (Circular Queue)* with two logical pointers: *in (rear)* and *out (front)*.
- The variable *in* points to the next free position in the buffer; out points to the first full position in the buffer.
- The buffer is empty when *in== out*;
- The buffer is full when *((in+ 1)% BUFFER_SIZE) == out*.
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
Solution is correct, but can only use BUFFER_SIZE-1 elements
```

2.10.2.2 Bounded Buffer

Producer process

```
item nextProduced;
while (true) {

/*  produce an item and put in nextProduced  */
    while (count == BUFFER_SIZE)
            ; // do nothing (Wait for buffer empty)
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Consumer process

```
item nextConsumed;
while (true)  {
    while (count == 0)
        ; // do nothing (wait for buffer full)
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    count--;
        /*  consume the item in nextConsumed
}
```

- The statements
  count++;
  count--; must be performed atomically.
- When producer produces an item: count++
- When Consumer consumes an item: count--

- Atomic operation means an operation that completes in its entirety without interruption.
- Both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.
- The statement "**count++**" may be implemented in machine language as:
  register1 = count
  register1 = register1 + 1
  count = register1
- The statement "**count--**" may be implemented as:
  register2 = count
  register2 = register2 – 1
  count = register2
- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.
- Consider this execution interleaving with "count = 5" initially:
  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}
- The value of **count** may be either 4 or 6, where the correct result should be 5.
- If several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a ***Race Condition***.

2.10.3 Race Condition

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

Suppose that two processes, P1 and P2, share the global variable a. At some point in its execution, P1 updates a to the value 1, and at some point in its execution, P2 updates a to the value 2. Thus, the two tasks are in a race to write variable a. In this example the "loser" of the race (the process that updates last) determines the final value of a.

Therefore Operating System Concerns of following things

1. The operating system must be able to keep track of the various processes
2. The operating system must allocate and deallocate various resources for each active process.
3. The operating system must protect the data and physical resources of each process against unintended interference by other processes.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

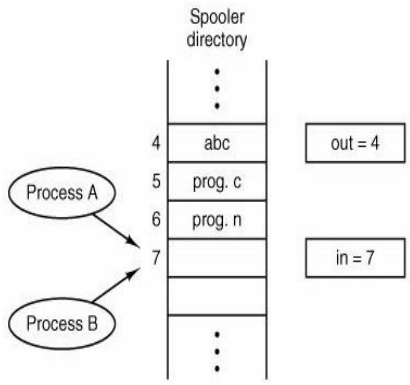
Process Interaction can be defined as

- Processes unaware of each other
- Processes indirectly aware of each other
- Processes directly aware of each other

Concurrent processes come into conflict with each other when they are competing for the use of the same resource.

Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of the other processes. There is no exchange of information between the competing processes.

The situation where two or more processes are reading or writing some shared data & the final results depends on who runs precisely when called race conditions are.
To see how inter process communication works in practice, let us consider a simple but common example, a print spooler. When a process wants to print a file, it enters the file name in a special spooler directory. Another process, the printer daemon, periodically checks to see if there are any files to be printed, and if there are, it prints them and removes their names from the directory.



Imagine that our spooler directory has a large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables,
out: which points to the next file to be printed
in: which points to the next free slot in the directory.
At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files to be printed). More or less simultaneously, processes A and B decide they want to queue a file for printing as shown in the fig.

Process A reads in and stores the value, 7, in a local variable called **next_free_slot**. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.
Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things.

Eventually, process A runs again, starting from the place it left off last time. It looks at **next_free_slot**, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes **next_free_slot + 1**, which is 8, and sets **in** to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

2.10.4 Avoiding Race Conditions:

2.10.4.1. Critical Section:

- n processes all competing to use some shared data (Changing Common Variables, Updating a Table, Writing a file and so on.)
- Each process has a code segment, called critical section, in which the shared data is accessed.

- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- 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.

2.10.4.2 Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress**. When no process is executing in its critical section and there exist some processes that wish to enter their critical section, must be permitted to enter into the critical section without delay.
3. **Bounded Waiting** - There is an upper bound on the number of times a process can enter it's critical section while another one is waiting (a.k.a. starvation)

Mutual Exclusion is some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

The difficulty above in the printer spooler occurs because process B started using one of the shared variables before process A was finished with it.

If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid race conditions. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.

2.10.4.3. Requirements for Mutual Exclusion

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its non-critical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

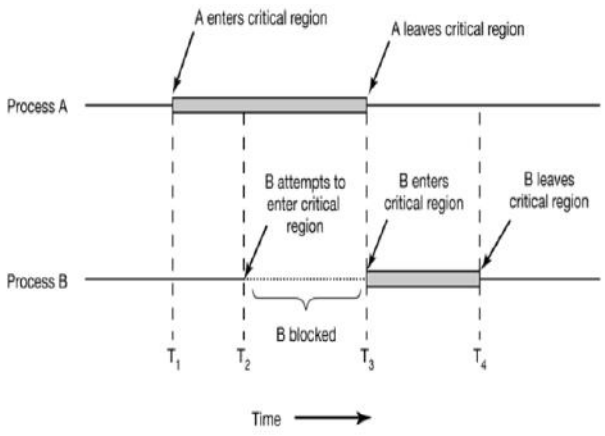


Figure 49: Mutual Exclusion Using Critical Regions

- Two general approaches are used to handle critical sections in operating systems are:
  - preemptive kernels and
  - non-preemptive kernels.
- A *preemptive 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.
- In preemptive kernels, it must be carefully designed to ensure that shared kernel data are free from race conditions.
- A preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.
- Furthermore, a preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before releasing the processor to waiting processes.

General structure of process

- General structure of process P_i (other process P_j)

```
do {
        entry section
                critical section
        exit section
                reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.
- Each process must request permission to enter its *critical section (CS)*.
- 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*.

2.10.4.4. Mutual Exclusion with busy waiting – Software Support

Software approaches can be implemented for concurrent processes that executes on a single processor or a multiprocessor machine with shared main memory.

Peterson's Solution (Two Task Solutions (Initial Attempts)

- Only 2 processes, P_0 and P_1. (*two process solution*)
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:

```
int turn;
Boolean flag[2]
```

- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section

- *flag[i] = true* implies that process P_i is ready!

Algorithm 1

Shared variables:
int turn**; initially **turn = 0
turn = i $\Rightarrow$ *P_i can enter its critical section*
Process P_i

```
do {
        while (turn != i); //busy waiting
                critical section
        turn = j;
                remainder section
} while (TRUE);
```

Satisfies mutual exclusion, but not progress.

Algorithm 2

- Shared variables
  boolean flag[2];
  initially flag [0] = flag [1] = FALSE
  flag [i] = TRUE $\Rightarrow$ *P_i wants to enter its critical section*
- Process P_i

```
do {
        flag[i] = TRUE;
                while (flag[j]);        //Busy Waiting
                critical section
        flag [i] = FALSE;
                remainder section
} while (TRUE);
```

- Satisfies mutual exclusion, but not progress requirements.

Algorithm 3 (Peterson's Algorithm)

- Combined shared variables of algorithms 1 and 2.
  initially flag [0] = flag [1] = false.
- Process P_i

```
do {
flag [i]:= true;
                turn = j;
                while (flag [j] and turn == j) ; //busy waiting
//critical section
flag [i] = false;
//remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Process 0	Process 1
i=0, j=1	i=1, j=0
flag[0] = TRUE turn = 1 - Lose processor here	
	flag[1] := TRUE turn := 0 check (flag[0] = TRUE and turn = 0) - Condition is true so Process 1 busy waits until it loses the processor
check (flag[1] = TRUE and turn = 1) - This condition is false because turn = 0 - No waiting in loop - Enters critical section	

Example 1

Example 2

EXAMPLE 1	
Process 0	Process 1
i = 0, j = 1	i = 1, j = 0
flag[0] := TRUE turn := 1 check (flag[1] = TRUE and turn = 1) - Condition is false because flag[1] = FALSE - Since condition is false, no waiting in while loop - Enter the critical section - Process 0 happens to lose the processor	
	flag[1] := TRUE turn := 0 check (flag[0] = TRUE and turn = 0) - Since condition is true, it keeps busy waiting until it loses the processor
- Process 0 resumes and continues until it finishes in the critical section - Leave critical section flag[0] := FALSE - Start executing the remainder (anything else a process does besides using the critical section) - Process 0 happens to lose the processor	
	check (flag[0] = TRUE and turn = 0) - This condition fails because flag[0] = FALSE - No more busy waiting - Enter the critical section

Example 3

EXAMPLE 3	
Process 0	Process 1
i=0, j=1	i=1, j=0
flag[0] = TRUE - Lose processor here	
	flag[1] = TRUE turn = 0 check (flag[0] = TRUE and turn = 0) - Condition is true so, Process 1 busy waits until it loses the processor
turn := 1 check (flag[1] = TRUE and turn = 1) - Condition is true so Process 0 busy waits until it loses the processor	
	check (flag[0] = TRUE and turn = 0) - The condition is false so, Process 1 enters the critical section

2.10.5. Mutual Exclusion – Hardware Support

Hardware approaches to mutual exclusion.

2.10.5.1. Interrupt Disabling

In a uniprocessor machine, concurrent processes cannot be overlapped; they can only be interleaved. Furthermore, a process will continue to run until it invokes an operating system service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the system kernel for disabling and enabling interrupts.

eg:

while (true) (
disable interrupts()
critical section enable interrupts()
) remainder

Because the critical section cannot be interrupted, mutual exclusion is guaranteed.

Advantages:

- It is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.

Disadvantages

1. It works only in a single processor environment.
2. Interrupts can be lost if not serviced promptly.
3. A process waiting to enter its critical section could suffer from starvation.

2.10.5.2. Lock Variables

- A single, shared, (lock) variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Drawbacks:
Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

2.10.5.3. Test and Set Instruction

It is special machine instruction used to avoid mutual exclusion.
TSL RX, LOCK (Test and Set Lock) that works as follows:
It reads the contents of the memory word LOCK into register RX and then stores a nonzero value at the memory address LOCK. The operations of reading the word and storing into it are guaranteed to be indivisible no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

The test and set instruction can be defined as follows:

```
boolean TestAndSet (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

Shared boolean variable lock, initialized to false

```
Solution:
    while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing
            //   critical section
        lock = FALSE;
            //   remainder section
    }
```

Process 1	Process 2
• Wants to set lock to TRUE • Lock is changed to True • Result comes back as FALSE so no busy waiting	
	• Wants to set lock to TRUE • It receives the result TRUE • Busy waits as long as Process 1 is in the critical section
• Leaves critical section • Sets lock to FALSE	

return false;The above function is carried out automatically.

Advantages

1.It is simple and easy to verify.
2.It is applicable to any number of processes.

```
enter_region:
TSL REGISTER,LOCK      [copy LOCK to register and set LOCK to 1
CMP REGISTER,#0        [was LOCK zero?
JNE enter_region       [if it was non zero, LOCK was set, so loop
RET                    [return to caller; critical region entered leave_region:
MOVE LOCK, #0          [store a 0 in LOCK RET                    [return to caller
```

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls enter_region, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls leave_region, which stores a 0 in LOCK. As with all solutions based on critical regions, the processes must call enter_region and leave_region at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.

Problems with mutual exclusion

The above techniques achieves the mutual exclusion using busy waiting. Here while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

Mutual Exclusion with busy waiting just check to see if the entry is allowed when a process wants to enter its critical region, if the entry is not allowed the process just sits in a tight loop waiting until it is

1. This approach waste CPU time
2. There can be an unexpected problem called priority inversion problem.

2.10.5.4. Swap Instruction

- Another type of hardware assistance for process synchronization.
- A special hardware instruction that does a complete swap (ordinarily three operations) **atomically**

- i.e., the complete swap is executed or no part of it
- given pass-by-reference parameters A and B, it swaps their values

Definition:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

- Shared boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- intuition: if lock is false, then a process can enter the critical section, and otherwise it can't
- Solution:

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key ); //Busy Waiting (If Lock = True)
                   // critical section
         lock = FALSE;
                  // remainder section
}
```

Bounded-Waiting Mutual Exclusion with TestAndSet()

```
do {
                waiting[i] = TRUE;
                key = TRUE;
                while (waiting[i] && key)
                        key = TestAndSet(&lock);
                waiting[i] = FALSE;
                        // critical section
                j = (i + 1) % n;
                while ((j != i) && !waiting[j])
                        j = (j + 1) % n;
                if (j == i)  lock = FALSE;
                else waiting[j] = FALSE;
                        // remainder section
        } while (TRUE);
```

1. To prove that the mutual exclusion requirement is met,
   - process P_i can enter its critical section only if either waiting [i] == false or key == false.
   - The value of key can become false only if the TestAndSet () is executed.
   - The first process to execute the TestAndSet () will find key== false; all others must wait.
   - The variable waiting [i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual-exclusion requirement.
2. To prove that the progress requirement is met,

- The arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false.
- Both allow a process that is waiting to enter its critical section to proceed.

3. To prove that the bounded-waiting requirement is met,
   - When a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n 1, 0, ... , i 1).
   - It designates the first process in this ordering that is in the entry section (waiting[j] ==true) as the next one to enter the critical section.
   - Any process waiting to enter its critical section will thus do so within n - 1 turns.

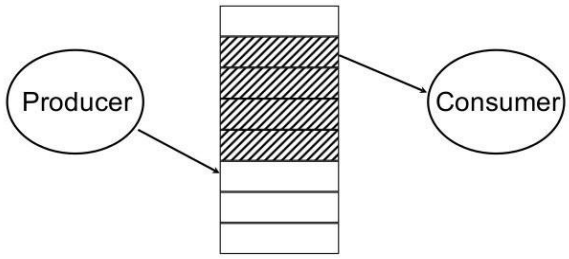
2.10.5.5. Sleep and Wakeup:

Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

Examples to use Sleep and Wakeup primitives:

Producer-consumer problem (Bounded Buffer):

Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.



Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty.
   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

```
#define N 100        /* number of slots in the buffer */
int count = 0;       /* number of items in the buffer */

void producer (void)
{
    int item;

    while (TRUE) {                  /* repeat forever */
        item = produce_item();      /* generate next item */
        if (count == N) sleep();    /* if buffer is full, go to sleep */
        insert_item(item);          /* put item in buffer */
        count = count + 1;          /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;
```

```
    while (TRUE) {                  /* repeat forever */
        if (count == 0) sleep();    /* if buffer is empty, got to sleep */
        item = remove_item();       /* take item out of buffer */
        count = count - 1;          /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);         /* print item */
    }
}
```

Figure 50: The producer-consumer problem with a fatal race condition

N = size of buffer
Count--> a variable to keep track of the no. of items in the buffer.

Producer's code:

The producers code is first test to see if count is N. If it is, the producer will go to sleep; if it is not the producer will add an item and increment count.

Consumer code:
It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.
Each of the process also tests to see if the other should be awakened and if so wakes it up.
This approach sounds simple enough, but it leads to the same kinds of race conditions as we saw in the spooler directory.

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. (Consumer is interrupted and producer resumed)
3. The producer creates an item, puts it into the buffer, and increases count.
4. Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer.
5. Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal** is lost.
6. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
7. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. For temporary solution we can use wakeup waiting bit to prevent wakeup signal from getting lost, but it can't work for more processes.

2.10.5.6.Semaphores

The solutions of the critical section problem represented in the section is not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool call a semaphore. A semaphore S is an integer variable that, a part from initialization, is an accessed two standard atomic operations: wait and signal. This operations were originally termed P

```
signal(S)
{
        S = S + 1;
}
```

The integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

In addition, in the case of the wait(S), the testing of the integer value of S (S0), and its possible modification (S: = S – 1), must also be executed without interruption.

Semaphores are not provided by hardware. But they have several attractive properties:

1. Semaphores are machine independent.
2. Semaphores are simple to implement.
3. Correctness is easy to determine.
4. Can have many different critical sections with different semaphores.
5. Semaphore acquire many resources simultaneously.

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
  - Can implement a counting semaphore S as a binary semaphore
  - Provides mutual exclusion

```
Semaphore S;    // initialized to 1
wait (S);
        Critical Section
signal (S);
```

Semaphores (Properties)

- Machine independent: no need to code at assembly level as in TSL
- Works with any no of processes
- Can have different semaphores for different critical section
- Simply binary semaphores or more than one if desired using a counting semaphore

Semaphore Implementation

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

Semaphore Implementation with no busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items
  - value (of type integer)

- pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue

Implementation of wait:

```
wait (S) {
    value--;
    if (value < 0) {
        add this process to waiting queue
        block(); }
}
```

Implementation of signal:

```
signal (S) {
        value++;
    if (value <= 0) {
        remove a process P from the waiting queue
        wakeup(P); }
}
```

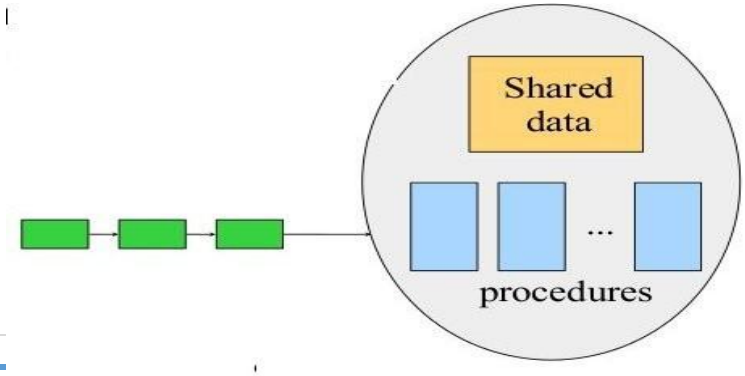
Drawback of Semaphore

1. They are essentially shared global variables.
2. Access to semaphores can come from anywhere in a program.
3. There is no control or guarantee of proper usage.
4. There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
5. They serve two purposes, mutual exclusion and scheduling constraints.

2.10.6 Monitors

In concurrent programming, a monitor is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.



With semaphores IPC seems easy, but suppose that the two downs in the producer's code were reversed in order, so mutex was decremented before empty instead of after it. If the buffer were completely full, the producer would block, with mutex set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on mutex, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock.

- A higher level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.
- This rule, which is common in modern object-oriented languages such as Java, was relatively unusual for its time,
- Section of code below illustrates a monitor written in an imaginary language, Pidgin Pascal.

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .    .    .
        end;
    end monitor;
```

2.10.6.1.Monitor with Signal

A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data. The characteristics of a monitor are the following:

1. The local data variables are accessible only by the monitor's procedures and not by any external procedure.
2. A process enters the monitor by invoking one of its procedures.
3. Only one process may De executing in the monitor at a time; any other process that has invoked the monitor is blocked, waiting for the monitor to become available.

A monitor supports synchronization by the use of condition variables that are contained Within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions:

- ***cwait (c)or wait():*** Suspend execution of the calling process on condition c. The monitor is now available for use by another process.
- ***csignal (c) or signal():*** Resume execution of some process blocked after a *cwait or wait* on the same condition)lf there are several such processes, choose one of them; if there is no such process, do nothing.

Monitor wait and signal operations are different from those for the semaphore. If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.

Although a process can enter the monitor by invoking any of its procedures, we can think of the monitor as having a single entry point that is guarded so that only one process may be in the monitor at a time. Other processes that attempt to enter the monitor join a queue of processes blocked waiting for monitor availability.

- Once a process is in the monitor, it may temporarily block itself on condition x by issuing ***cwait (x) or wait();*** it is then placed in a queue of processes waiting to reenter the monitor when the condition changes, and resume execution at the point in its program following the ***cwait(x)*** or ***wait()*** call.

- If a process that is executing in the monitor detects a change in condition variable x, it issues ***csignal (x) or signal(),*** which alerts the corresponding condition queue that the condition has changed.
- A producer can add characters to the buffer only by means of the procedure append inside the monitor; the producer does not have direct access to buffer.
- The procedure first-checks the condition not full to determine if there is space available in the buffer. If not, the process executing the monitor is blocked on that condition.

2.10.7. Classical IPC (Synchronization) Problems

There are various synchronization problems of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme.

1. Bounded buffer problem
2. Readers and Writers Problem
3. Dining-Philosophers Problem
4. Sleeping barber Problem

1.10.7.1. Bounded buffer problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

- The structure of the producer 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 consumer process*

```
do {
```

```
        wait (full);
        wait (mutex);
              // remove an item from buffer to nextc
        signal (mutex);
        signal (empty);

              // consume the item in nextc
    } while (TRUE);
```

Semaphore

- Solve producer-consumer problem
  - Full: counting the slots that are full; initial value 0
  - Empty: counting the slots that are empty, initial value N
  - Mutex: prevent access the buffer at the same time, initial value 0 (**binary semaphore**)
  - Synchronization/mutual exclusion

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                          /* decrement empty count */
          down(&mutex);                          /* enter critical region */
          insert_item(item);                     /* put new item in buffer */
          up(&mutex);                            /* leave critical region */
          up(&full);                             /* increment count of full slots */
     }
}
```

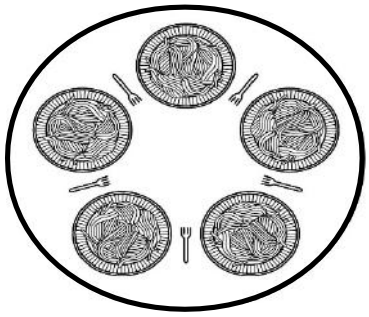
```
void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                           /* decrement full count */
          down(&mutex);                          /* enter critical region */
          item = remove_item( );                 /* take item from buffer */
          up(&mutex);                            /* leave critical region */
          up(&empty);                            /* increment count of empty slots */
          consume_item(item);                    /* do something with the item */
     }
}
```

Figure 51: Consumer

2.10.7.2. Dining philosopher's problems

There are N philosophers sitting around a circular table eating spaghetti and discussing philosophy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.



- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock

The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible.

One idea is to instruct each philosopher to behave as follows:

- Think until the left fork is available; when it is, pick it up
- Think until the right fork is available; when it is, pick it up
- Eat
- Put the left fork down
- Put the right fork down
- Repeat from the start

This solution is incorrect: it allows the system to reach deadlock. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, and picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation.

The solution presented below is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3.

```
#define N               5   /* number of philosophers */
#define LEFT            (i+N-1)%N /* number of i's left neighbor */
#define RIGHT           (i+1)%N /* number of i's right neighbor */
#define THINKING        0   /* philosopher is thinking */
#define HUNGRY          1   /* philosopher is trying to get forks */
#define EATING          2   /* philosopher is eating */
typedef int semaphore;      /* semaphores are a special kind of int */
int state[N];               /* array to keep track of everyone's state */
semaphore mutex = 1;        /* mutual exclusion for critical regions */
semaphore s[N];             /* one semaphore per philosopher */

void philosopher (int i)    /* i: philosopher number, from 0 to N-1 */
{
```

```
    while (TRUE) {          /* repeat forever */
        think();            /* philosopher is thinking */
        take_forks(i);      /* acquire two forks or block */
        eat();              /* yum-yum, spaghetti */
        put_forks(i);       /* put both forks back on table */
    }
}

void take_forks(int i)      /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);           /* enter critical region */
    state[i] = HUNGRY;      /* record fact that philosopher i is hungry */
    test(i);                /* try to acquire 2 forks */
    up(&mutex);             /* exit critical region */
    down(&s[i]);            /* block if forks were not acquired */
}

void put_forks(i)           /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);           /* enter critical region */
    state[i] = THINKING;    /* philosopher has finished eating */
    test(LEFT);             /* see if left neighbor can now eat */
    test(RIGHT);            /* see if right neighbor can now eat */
    up(&mutex);             /* exit critical region */
}

void test(i)                /* i: philosopher number, from 0 to N-1 */
{
  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
EATING) {
    state[i] = EATING;
    up(&s[i]);
  }
}
```

Figure 52: A solution to the dining philosophers' problem.

2.10.7.3. Readers Writer problems

The dining philosophers' problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database (Courtois et al., 1971). Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader. The question is how do you program the readers and the writers? One solution is shown below.

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {              /* repeat forever */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc + 1;            /* one reader more now */
        if (re == 1) down(&db);     /* if this is the first reader… */
        up{&mutex);             /* release exclusive access to 'rc' */
```

```
        read_data_base();       /* access the data */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc - 1;             /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader… */
        up(&mutex);             /* release exclusive access to 'rc' */
        use_data_read();        /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {              /* repeat forever */
        think_up_data();        /* noncritical region */
        down(&db);              /* get exclusive access */
        write_data_base();      /* update the data */
        up(&db);                /* release exclusive access */
    }
}
```

Figure 53: A solution to the readers and writers problem.

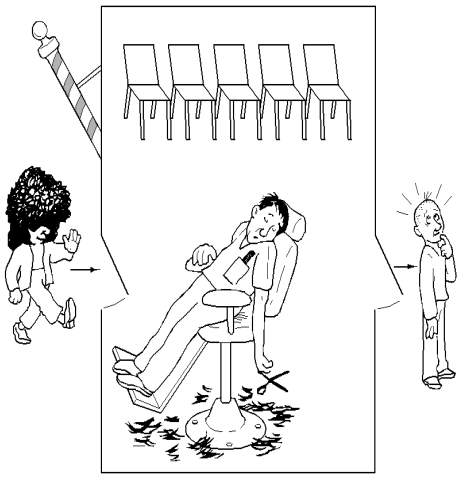
In this solution, the first reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. A third and subsequent readers can also be admitted if they come along.

Now suppose that a writer comes along. The writer cannot be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.

To prevent this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance.

2.10.7.4. The Sleeping Barber Problem



Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and *n* chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in Fig. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions. This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.

This solution uses three semaphores: *customers*, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), *barbers*, the number of barbers (0 or 1) who are idle, waiting for customers, and *mutex*, which is used for mutual exclusion. We also need a variable, *waiting*, which also counts the waiting customers. It is essentially a copy of *customers*. The reason for having *waiting* is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

Our solution is shown in Fig. below. When the barber shows up for work in the morning, he executes the procedure *barber,* causing him to block on the semaphore *customers* because it is initially 0. The barber then goes to sleep. He stays asleep until the first customer shows up.

```
#define CHAIRS 5               /* # chairs for waiting customers */
typedef int semaphore;         /* use your imagination */
semaphore customers = 0;       /* # of customers waiting for service */
semaphore barbers = 0;         /* # of barbers waiting for customers */
semaphore mutex = 1;           /* for mutual exclusion */
int waiting = 0;               /* customers are waiting (not being cut) */

void barber(void)
{
    white (TRUE) {
        down(&customers);      /* go to sleep if # of customers is 0 */
        down(&mutex);          /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers);          /* one barber is now ready to cut hair */
        up(&mutex);            /* release 'waiting' */
        cut_hair();            /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);              /* enter critical region */
    if (waiting < CHAIRS) {    /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers);        /* wake up barber if necessary */
```

```
        up(&mutex);             /* release access to 'waiting' */
        down(&barbers);         /* go to sleep if # of free barbers is 0 */
        get_haircut();          /* be seated and be serviced */
    } else {
        up(&mutex);             /* shop is full; do not wait */
    }
}
```

Figure 54: A solution to the sleeping barber problem.

When a customer arrives, he executes *customer*, starting by acquiring *mutex* to enter a critical region. If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released *mutex*. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases *mutex* and leaves without a haircut.

If there is an available chair, the customer increments the integer variable, *waiting*. Then he does an up on the semaphore *customers*, thus waking up the barber. At this point, the customer and barber are both awake. When the customer releases *mutex*, the barber grabs it, does some housekeeping, and begins the haircut.

When the haircut is over, the customer exits the procedure and leaves the shop. The barber loops, however, to try to get the next customer. If one is present, another haircut is given. If not, the barber goes to sleep.

2.11. DEADLOCK: Introduction

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 wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes. This situation is called deadlock.

If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly.

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.

There are two types of resources:
Preemptable - A Preemptable resources is one that can be taken away from the process owing it with no ill effect. Memory is an example of preemptable resources.

Non-preemptable - A non-preemptable resources in contrast is one that cannot be taken away from its current owner without causing the computation to fail. Examples are CD-recorder and Printers. If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD. CD recorders are not preemptable at any arbitrary moment.

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

1. Request: If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. Use: The process can operate on the resource.
3. Release: The process releases the resource

2.12. What is Deadlock?

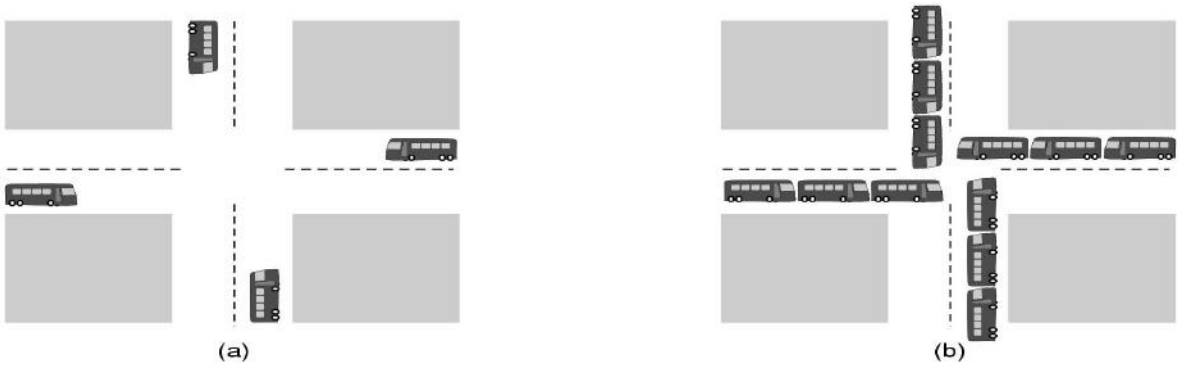
In Computer Science a set of process is said to be in deadlock if each process in the set is waiting for an event that only another process in the set can cause. Since all the processes are waiting, none of them will ever cause any of the events that would wake up any of the other members of the set & all the processes continue to wait forever.

Example 1:

- Two process A and B each want to record a scanned document on a CD.
- A requests permission to use Scanner and is granted.
- B is programmed differently and requests the CD recorder first and is also granted.
- Now, A ask for the CD recorder, but the request is denied until B releases it. Unfortunately, instead of releasing the CD recorder B asks for Scanner. At this point both processes are blocked and will remain so forever. This situation is called Deadlock.

Example 2

Bridge Crossing Example:



- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ==>no one goes west

2.12.1. Starvation vs. Deadlock

- ***Starvation:*** thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- ***Deadlock:*** circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2 Thread B owns Res 2 and is waiting for Res 1
- Deadlock ==> Starvation but not vice versa
- Starvation can end (but doesn't have to)
- Deadlock can't end without external intervention

2.13. Deadlock Characterization

In deadlock, processes never finish executing and system resources are tied up, preventing other jobs from ever 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***: There must exist a process that is holding at least one resource and is 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:** There must exist a set {P0, P1, ..., Pn } of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …., Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

2.14. Resource-Allocation Graph (Modeling deadlock)

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. The set of vertices V is partitioned into two different types of nodes P = {P_1, P_2, …P_n} the set consisting of all the active processes in the system; and R = {R_1, R_2, …, R_1}, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$, it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a request edge; a directed edge $R_j \rightarrow$ Pi is called an assignment edge.

When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the, resource it releases the resource, and as a result the assignment edge is deleted.

Definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains the cycle, then a deadlock must exist.

If each resource type has several instances, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resources types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

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

V is partitioned into two types:

- P = {P_1, P_2, …, P_n}, the set consisting of all the processes in the system.

- $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system. request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

If each resource type has several instance, then a cycle does not necessarily imply that a deadlock incurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

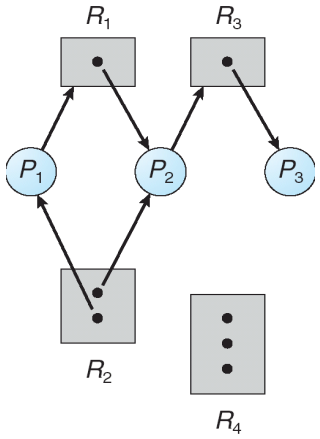


Fig: Resource allocation graph

The sets P, K and E:
P == $\{P_1, P_2, P_3\}$
R== $\{R_1, R_2, R_3, R_4\}$
E == $\{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

Resource instances:

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R4

Process states:

- Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
- Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
- Process P_3 is holding an instance of R_3.

Suppose that process P_3 requests an instance of resource type R_2 Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph. At this point, two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

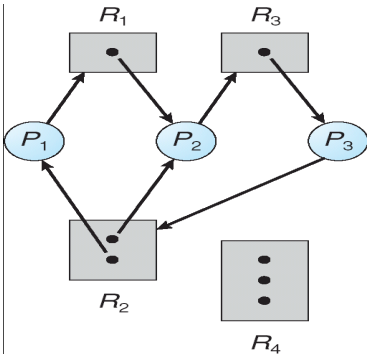
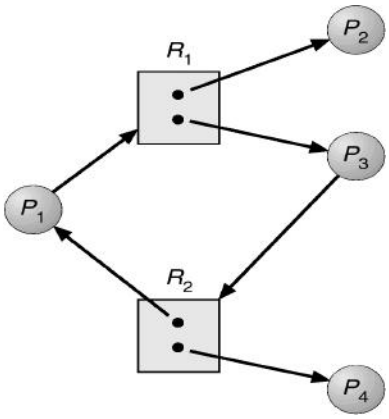


Figure 55: Resource Allocation Graph with Deadlock

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3,on the other hand, is waiting for either process P1 or process P2 to release resource R2. In addition, process PI is waiting for process P2 to release resource R1.



We have a cycle in Fig P1→ R_1→P3→R_2→P_1

However, there is no deadlock.

Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

Figure 56: Resource Allocation Graph with cycle but no deadlock

Note:

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, deadlock possible.
- If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state.
- If there is a cycle, then the system may or may not be in a deadlocked state.
- This observation is important when we deal with the deadlock problem.

Deadlock Modeling

Deadlock handling strategies:

1. Just Ignore the problem altogether.
2. Detection and recovery. Let deadlocks occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.

The Ostrich Algorithm (Ignore deadlock)

- "Stick your head in the sand and pretend there is no problem".
- Mathematicians find it totally unacceptable and say that deadlocks must be prevented.
- Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is.

2.15. Method for Handling Deadlock //Detection

There are three different methods for dealing with the deadlock problem:

- We can use a protocol to ensure that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state and then recover.
- We can ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.

Handling Deadlock

1. Deadlock prevention
2. Deadlock avoidance
3. Deadlock detection
4. Recovery from deadlock

Deadlock avoidance, on the other hand, 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, we can decide for each request whether or not the process should wait. Each request requires that the system consider the resources

currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlock state yet has no way of recognizing what has happened.

2.15.1. Deadlock Prevention

For a deadlock to occur, each of the four necessary-conditions must hold. By ensuring that at least on one these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. Denying the Mutual Exclusion Condition.
2. Denying the Hold and Wait Condition.
3. Denying the No Preemption Condition.
4. Denying the Circular Wait Condition.

2.15.1.1. Mutual Exclusion

The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources, on the other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock.

Principle:

- Avoid assigning resource when not absolutely necessary
- As few processes as possible actually claim the resource

2.15.1.2. Hold and Wait

1. When whenever a process requests a resource, it does not hold any other resources. One protocol that be used requires each process to request and be allocated all its resources before it begins execution.

2. An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated here are two main disadvantages to these protocols. *First, resource utilization may be low*, since many of the resources may be allocated but unused for a long period. *Second, starvation is possible.*

Problems

- May not know required resources at start of run.
- Also ties up resources other processes could be using.
- Resources will not be used optimally with this approach.

Variation:

- Process must give up all resources.
- Then request all immediately needed.

Example

- We consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.

First Method:

- If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer.
- It will hold the printer for its entire execution, even though it needs the printer only at the end.

Second Method:

- Allows the process to request initially only the DVD drive and disk file.
- It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file.
- The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

2.15.1.3.No Preemption

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

2.15.1.4.Circular Wait

Circular-wait condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Let R = $\{R_1, R_2, ..., R_n\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R → N, where N is the set of natural numbers.

Example:

- Let R = $\{ R_1, R_2, ... , R_m\}$ be the set of resource types.
- We assign to each resource type a unique integer number.
- Formally, we define a one-to-one function F:R→ N, where N is the set of natural numbers.
- For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:
  F (tape drive) = 1
  F (disk drive) = 5
  F (printer) = 12
- ***Protocol to prevent deadlocks***
  - Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type -say, R_i. After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$
  - For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

- Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$.
- It must also be noted that if several instances of the same resource type are needed, a single request for all of them must be issued.
- From these protocols we can say that the circular-wait condition cannot hold.

Demonstration of circular wait exists (proof by contradiction).

- Let the set of processes involved in the circular wait be $\{P_0, P_1, \ldots, Pn\}$, where P_i is waiting for a resource R_i, which is held by process P_{i+1}. (Modulo arithmetic is used on the indexes, so that P_n is waiting for a resource R_n held by P_0)
- Then, since process P_{i+1} is holding resource R_i while requesting resource R_{i+1} , we must have $F(R_i) < F(R_{i+1})$ for all i. But this condition means that $F(R_0) < F(R_1) < \ldots.. < F(R_n) < F(R_0)$.
- By transitivity, $F(R_0) < F(R_0)$, which is impossible.
- Therefore, there can be no circular wait.

2.15.2. Deadlock Avoidance

- Requires that the system has some additional a priori information available
- Deadlock avoidance is achieved by being careful at the time of resources allocation.
- The system must be able to decide whether granting a resource is safe or not, and only make the allocation when it is safe.
- Simplest and most useful model: each process declare the maximum number of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

2.15.2.1. Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes $<P_1, P_2, ..P_n>$ is a safe sequence for the current allocation state if, for each P_i the resources that P_j can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with j <i. In this situation, if the resources that process Pi needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

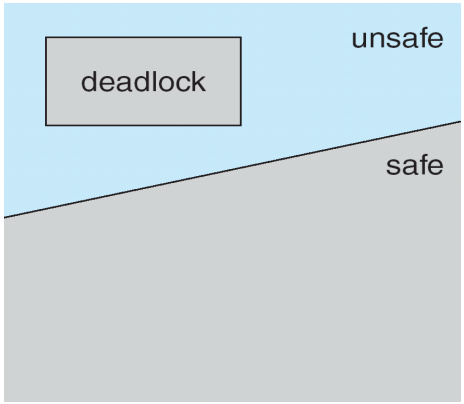


Figure 57: Safe, Unsafe & Deadlock State

If no such sequence exists, then the system state is said to be unsafe.

Basic Facts:

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

Let's consider that we have altogether 10 resources for 3 processes

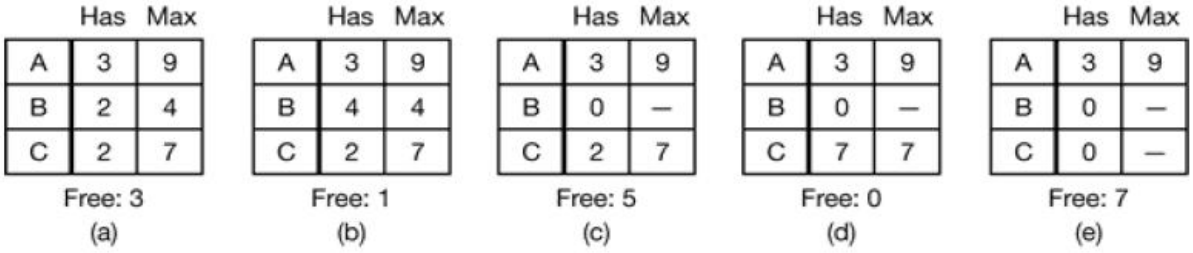


Figure 58: Demonstration that the figure in (a) is safe state

Sequence of run:

1. Process B take 2 resource and leads to figure (b). (Free=1)
2. Process B completes, obtain (c). (Free = 5)
3. Scheduler run process C (Take 5 resource) and leading to (d). (Free= 0)
4. When process completes, get (e). (Free=7)
5. Now process A can get 6 instances.
6. Thus (a) is safe state.

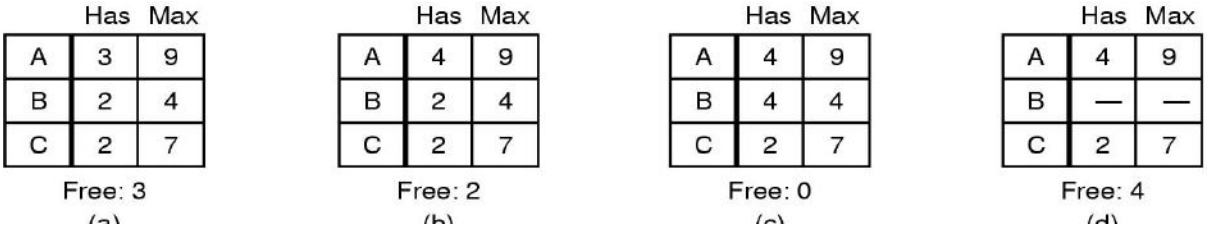


Figure 59: demonstration that the figure in (b) is not safe

Note that: *An unsafe state is not a deadlocked state.* System can run for a while. From a safe state, the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Resource-Allocation Graph Algorithm

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

2.15.2.2. Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource- allocation system with multiple instances of each resource type. The deadlock- avoidance algorithm that we

describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm.

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

Data Structures for the Banker's Algorithm (Multiple Resource)

- Let n = number of processes, and m = number of resource types
- ***Available****:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- ***Max****:* n x m matrix. If $Max\ [i,j] = k$, then process P_i may request at most k instances of resource type R_j
- ***Allocation****:* n x m matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j
- ***Need****:* n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

  $Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$.

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively.
   a. Initialize: $Work = Available$
   b. $Finish[i] = false$ for $i = 0, 1, \ldots, n\text{-}1$.
2. Find an i such that both:
   i. $Finish[i] = false$
   ii. $Need_i \leq Work$
3. If no such i exists, go to step 4.
   a. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2.
4. If $Finish\ [i]$ == true for all i, then the system is in a safe state.

Resource-Request Algorithm for Process P_i

$Request$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   a. $Available = Available - Request;$
   b. $Allocation_i = Allocation_i + Request_i;$
   c. $Need_i = Need_i - Request_i;$

   If safe ⇒ the resources are allocated to Pi.
   If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

Consider a system with five processes P0 through P4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken

Process	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

1. What will be the content of the need Matrix?
2. Is the system in safe state? If yes, then what is the safe sequence?

1. Need [i,j]= Max [i,j] - Allocation[i,j]
   Content of Need Matrix is

	A	B	c
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Applying Safety algorithms
For Pi if $Need_i$ <= Available, then P_i is in Safe sequence,
Available = Available + $Allocation_i$

For P0, need0=7, 4, 3
Available = 3, 3, 2
==> Condition is false, So P0 must wait.

For P1, need1= 1, 2, 2
Available=3, 3, 2
need1< Available
So P1 will be kept in safe sequence. & Available will be updated as:
Available= 3, 3, 2 + 2, 0, 0 = 5, 3, 2

For P2, need2= 6, 0, 0
Available = 5, 3, 2
==> Condition is again false, so P2 also have to wait.

For P3, need3= 0, 1, 1
Available= 5, 3, 2
==> Condition is true, P3 will be in safe sequence.
Available = 5, 3, 2 + 2, 1, 1 = 7, 4, 3

For P4, need4= 4, 3, 1
Available = 7, 4, 3
==> Condition $Need_i$ <= Available is true, so P4 will be in safe sequence
Available = 7, 4, 3 + 0, 0, 2 = 7, 4, 5

Now we have two processes P0 and P2 in waiting state. Either P0 or Pl can be chosen. Let us take P2 whose need = 6, 0, 0

Available = 7, 4, 5

Since condition is true, P2 now comes in safe state leaving the

Available = 7, 4, 5 + 3, 0, 2 = l0, 4, 7

Next P0 whose need = 7, 4, 3
Available = l0, 4, 7
Since condition is true PO also can be kept in safe state.
So system is in safe state & the safe sequence is <Pl, P3, P4, P2, P0>

2.15.3.Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.
- If in Resource Allocation Graph (RAG), every resource has only one instance (or single instance), then we define a deadlock detection algorithm that uses a variant of the RAG and is called **wait-for graph.**
- We can get this graph from RAG by **removing the nodes of type resource and collapsing the appropriate edge.** In this situation, if the wait-for graph has any cycle then there is a deadlock in the system.
- To detect deadlocks, the system needs to maintain the wait-for graph and to periodically invoke an algorithm that searches for a cycle in the graph.
- The complexity of this algorithm will be O (n^2) where n is the number of vertices in the graph.

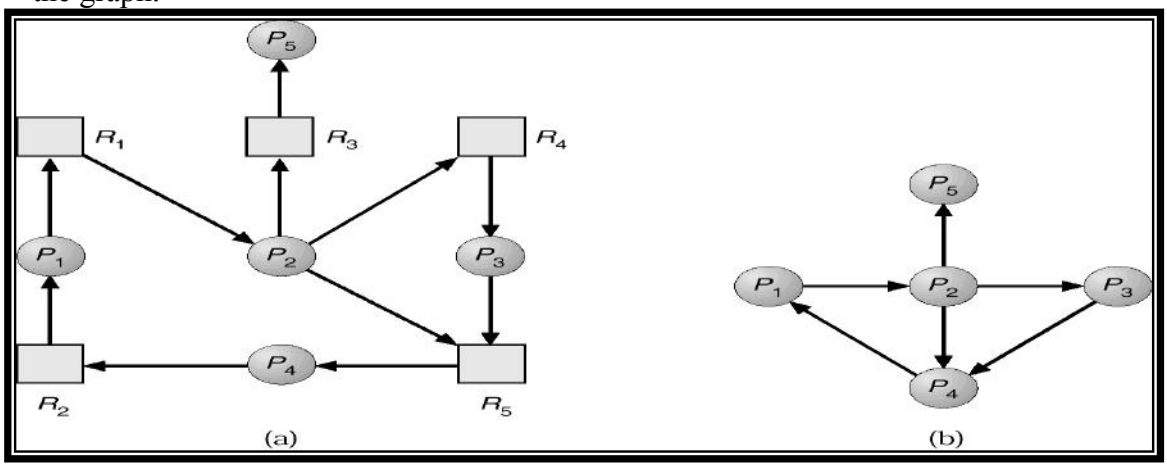


Figure 60: (a): Resource Allocation Graph Fig (b): Corresponding Wait for Graph

2.15.3.1. Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this

graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

2.15.3.2. Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.

The algorithm used are:

- Available: A vector of length m indicates the number of available resources of each type.
- Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.
- Request: An n x m matrix indicates the current request of each process. If Request [i, j] = k, then process P, is requesting k more instances of resource type R_j.

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then
   Finish[i] = false; otherwise, *Finish*[i] = *true*.
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$
   If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

2.15.4.Detection-Algorithm Usage

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

2.15.5. Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has spurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

2.15.5.1. Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- Abort all deadlocked processes: This method clearly will break the dead – lock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed.
- Abort one process at a time until the deadlock cycle is eliminated: This method incurs considerable overhead, since after each process is aborted a deadlock-detection algorithm must be invoked to determine whether a processes are still deadlocked.
- In which order should we choose to abort?
    i. Priority of the process
    ii. How long process has computed, and how much longer to completion
    iii. Resources the process has used
    iv. Resources process needs to complete
    v. How many processes will need to be terminated
    vi. Is process interactive or batch?

2.15.5.2. Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until he deadlock cycle is broken.

The three issues are considered to recover from deadlock

1. Selecting a victim
2. Rollback
3. Starvation

2.15.5.3. Recovery through rollback

- Checkpoint a process periodically *(check pointing a process means that its state is written to a file so that it can be restarted later).*
- To do the recovery, a process is rolled back to a point in time before it acquired some other resource by starting one of its earlier check-points.
- That is, restart the process if it is found deadlocked.

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***. The shared-memory method requires communicating processes through the use of these shared variables. In a shared-memory system, the responsibility for providing communication rests with the application programmers: the operating system needs to provide only the shared memory. The responsibility for providing communication may rest with the operating system itself. These two schemes are not mutually exclusive and can be used simultaneously within a single operating system.
- 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 many 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.
- Critical section is a code that only one process at a time can be executing. Critical section problem is design an algorithm that allows at most one process into the critical section at a time, without deadlock. Solution of the critical section problem must satisfy mutual exclusion, progress, bounded waiting.

- Semaphore is a synchronization variable that tasks on positive integer values. Binary semaphore are those that have only two values 0 and 1. Semaphores are not provided by hardware. Semaphore is used to solve critical section problem.
- A monitor is a software module consisting of one or more procedures, an initialization sequence and local data. Components of monitors are shared data declaration, shared data initialization, operations on shared data and synchronization statement.
- Various synchronization problems (such as the bounded-buffer problem, the readers-writers problem, sleeping barber problem and the dining-philosophers problem) are important mainly because they are examples of a large class of concurrency-control problems.
- A deadlocked state occurs when two or more processes are waiting indefinitely for an event that can be caused only one of the waiting processes. There are three principal methods for dealing with deadlocks:
    - Use some protocol to prevent or avoid deadlocks, entering that the system will never enter a deadlocked state.
    - Allow the system to enter a deadlocked state, detect it, and then recover.
    - Ignore the problem altogether and pretend that deadlocks never occur in the system.
- A deadlock can occur only if four necessary conditions hold simultaneously in the system: mutual exclusion, hold and wait, no preemption, Circular wait.
- To prevent deadlocks, we can ensure that at least one of the necessary conditions never holds.
- A method for avoiding deadlocks, rather than preventing them, requires that the operating system have a priori information about how each process will utilize system resources.
- The banker's algorithm, for example, requires a priori information about the maximum number of each resource class that each process may request.
    - Using this information, we can define a deadlock avoidance algorithm.
- If a system does not employ a protocol to ensure that deadlocks will never occur, then a detection-and-recovery scheme may be employed.
- A deadlock detection algorithm must be invoked to determine whether a deadlock has occurred.
- If a deadlock is detected, the system must recover either by terminating some of the deadlocked processes or by preempting resources from some of the deadlocked processes.
- Where preemption is used to deal with deadlocks, three issues must be addressed:
    - Selecting a victim, rollback, and starvation.
- In a system that selects victims for rollback primarily on the basis of cost factors, starvation may occur, and the selected process can never complete its designated task.

CHAPTER 3

MEMORY MANAGEMENT

Memory Management

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address.

A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use the process may be moved between disk and memory during its execution. The collection of processes on the disk that are waiting to be brought into memory for execution forms the input queue. i.e. selected one of the process in the input queue and to load that process into memory. We can provide protection by using two registers, usually a base and a limit, as shown in Figure 61 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).

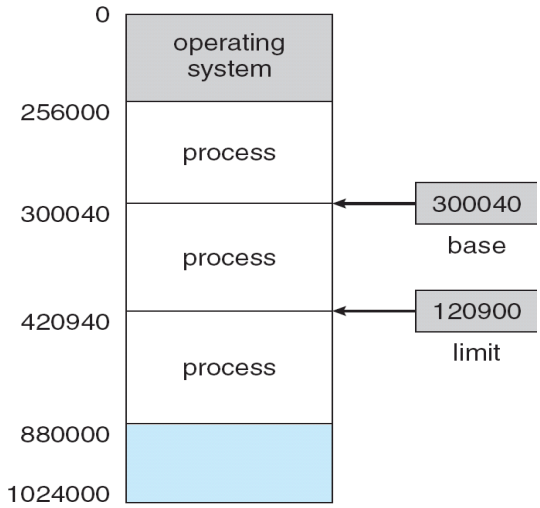


Figure 61: A base and limit register define a logical address space.

The binding of instructions and data to memory addresses can be done at any step along the way:

- ***Compile time***: If it is known at compile time where the process will reside in memory, then absolute code can be generated.
- ***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.
- ***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.

Dynamic Loading

Better memory-space utilization can be done by dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed.

The advantage of dynamic loading is that an unused routine is never loaded.

Most 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. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. With dynamic linking, a stub is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routing.

The entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. So that a process can be larger than the amount of memory allocated to it, a technique called overlays is sometimes used. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space that was occupied previously by instructions that are no longer needed.

Example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table

1, and common support routines used by both pass 1 and pass 2.

Let us consider

Pass1	70K
Pass 2	80K
Symbol table	20K
Common routines	30K

To load everything at once, we would require 200K of memory. If only

150K is available, we cannot run our process. But pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2.

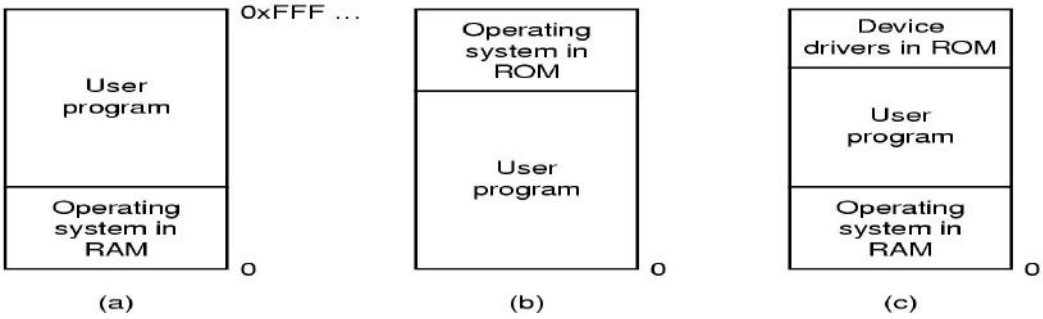
We add an overlay driver (10K) and start with overlay A in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass 2. Overlay A needs only 120K, whereas overlay B needs 130K

As in dynamic loading, overlays do not require any special support from the operating system.

Memory Management Basics

- Don't have infinite RAM
- Do have a memory hierarchy-
  - Cache (fast)
  - Main(medium)
  - Disk(slow)
- Memory manager has the job of using this hierarchy to create an **abstraction** (illusion) of easily accessible memory
- Two important memory management function:
  - **Sharing**
  - **Protection**

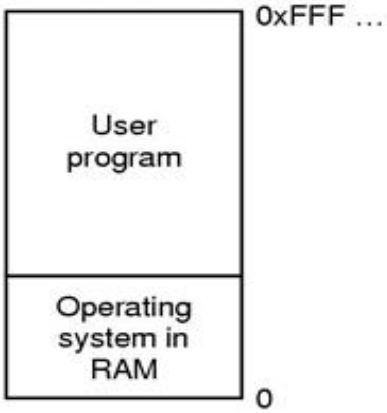
3.3.1 Monoprogramming Model



- Only one program at a time in main memory;
- Can use whole of the available memory.
- Bug in user program can trash the OS (a and c)
- Second on some embedded systems
- Third on MS-DOS (early PCs) -part in ROM called BIOS
- Since only one program or process resides in memory at a time, so sharing and protection is not an issue.
- However, the protection of OS program from the user code is must otherwise the system will crash down.

This protection is done by a special hardware mechanism such as a dedicated register, called as a **Fence Register.** Also called Limit Register.

- The Fence Register is set to highest address occupied by the OS code.
- A memory address generated by user program to access certain memory location is first compared with the fence register's content.
- If the address generated is below the fence, it will be trapped and denied permission.
- Since the modification of fence register is considered as a privileged operation, therefore, only OS is allowed to make any changes to it.



3.3.2. Address Binding (Relocation)

- Our program is stored in RAM only
- A program has to go through these three phases:
- Compilation, Loading and Execution
- The problem that arises is that where should an OS store the results of programs after execution.
- A user specifies in his instruction where to store the result.
    - i.e. x=(a+b)× (a-c)
- Such address given by the user, like x, are called symbolic or logical address.
- These addresses need to be mapped to real physical addresses in memory.
- The mechanism is called **address binding.**

3.3.3. 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 is commonly referred to as a physical address.

The compile-time and load-time address-binding schemes result in an environment where the logical and physical addresses are the same. The execution- time address-binding scheme results in an environment where the logical and physical addresses differ. In this case, we usually refer to the logical address as a virtual address. The set of all logical addresses generated by a program is referred to as a logical address space; the set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The run-time mapping from virtual to physical addresses is done by the memory management unit (MMU), which is a hardware device.

The base register is called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 13000, then an attempt by the user to address location 0 dynamically relocated to location 14,000; an access to location 347 is mapped to location 13347. The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes.

The user program never sees the real physical addresses. The program can create a pointer to location 347 store it memory, manipulate it, compare it to other addresses all as the number 347.

The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addressed Logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R). The user generates only logical addresses.

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

Program Relocation

- Refers to the ability to load and execute a given program into an arbitrary place in the memory.
- Relocation is a mechanism to convert the logical address into a physical address.
- To do this, there is a special register in CPU called relocation register.
- Every address used in the program is relocated as:
  effective physical address= Logical address + Contents of Relocation register
- MMU is a special hardware which performs address binding, uses relocation scheme.

Memory Management Unit (MMU)

- MMU generates physical address from virtual address provided by the program.
- MMU maps virtual addresses to physical addresses and puts them on memory bus
- Two basic types of relocation:

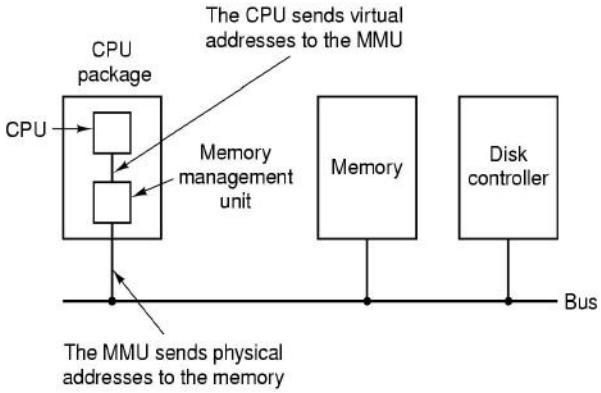


Figure 62: Memory management Unit

  - **Static Relocation:** Formed during the loading of the program into memory by a loader.
  - **Dynamic Relocation:** mapping from the virtual address space to physical address is performed at execution-time.

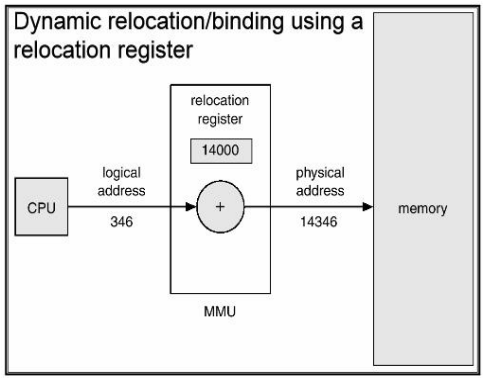


Figure 63: Dynamic relocation using relocation register

Protection

- Providing security from unauthorized usage of memory.
- OS can protect the memory with the help of **base and limit registers**.
- **Base register** consist of the starting address of the next process, the **limit register** specifies the boundary of that job.
- That is why the limit register is also called a **fencing register.**

  Fig: Hardware protection mechanism

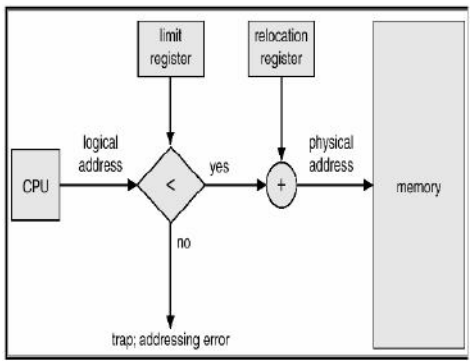


Figure 64: Memory protection using relocation register

3.1.1. Memory Allocation Techniques

- Two types:
  - Contiguous Storage Allocation
    - Fixed Partition Allocation
    - Variable Partition Allocation
  - Non-Contiguous
    - Paging
    - Segmentation

3.1.1.1. Contiguous Storage Allocation

- Main memory must accommodate both the operating system and the user processes
- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory
  - User processes then held in high memory
- In this case memory protection must be done.
- *Memory protection* means protecting the operating system from user processes and protecting user processes form one another.
- Memory protection can be done by using *relocation register*.
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data.
  - *Base register* contains value of smallest physical address.
  - Limit register contains range of logical addresses – each logical address must be less than the limit register.
  - MMU maps logical address *dynamically*
- A memory resident program occupies a single contiguous block of physical memory. The memory is partitioned into block of different sizes to accommodate the programs. The partitioning may be:
  - Fixed Partition allocation
  - Variable Partition allocation

Fixed Partition allocation/Multiprogramming with fixed partition

- In multiprogramming environment, several programs reside in primary memory at a time and the CPU passes its control rapidly between these programs.
- One way to support multiprogramming is to divide the main memory into several partitions each of which is allocated to a single process.

Equal-size partitions

- any process whose size is less than or equal to the partition size can be loaded into an available partition
- if all partitions are full, the operating system can swap a process out of a partition
- a program may not fit in a partition. The programmer must design the program with overlays

- **Disadvantage**:
  - Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called internal fragmentation.
  - Depending on how and when the partitions are created, there may be two types of partitioning:

- Static partitioning
- Dynamic partitioning

Static Vs Dynamic Partitioning

- **Static Partitioning**: implies that the division of memory into number of partitions and its size is made in the beginning prior to the execution of user programs and remains fixed thereafter.
- *Dynamic Partitioning*: the size and the number of partitions are decided during the execution time by the OS.

Operation Principle:

- We divide the memory into several fixed size partitions where each partition will accommodate only one program for execution.
- The number of programs (i.e. degree of multiprogramming) residing in the memory will be bound by the number of partitions.
- When a program terminates, that partition is freed for another program waiting in a queue.

Placement Algorithm with fixed Partitions

- Equal-size partitions
  - Any process whose size is less than or equal to the partition size can be loaded into an available partition
  - if all partitions are full, the operating system can swap a process out of a partition
  - A program may not fit in a partition. The programmer must design the program with overlays

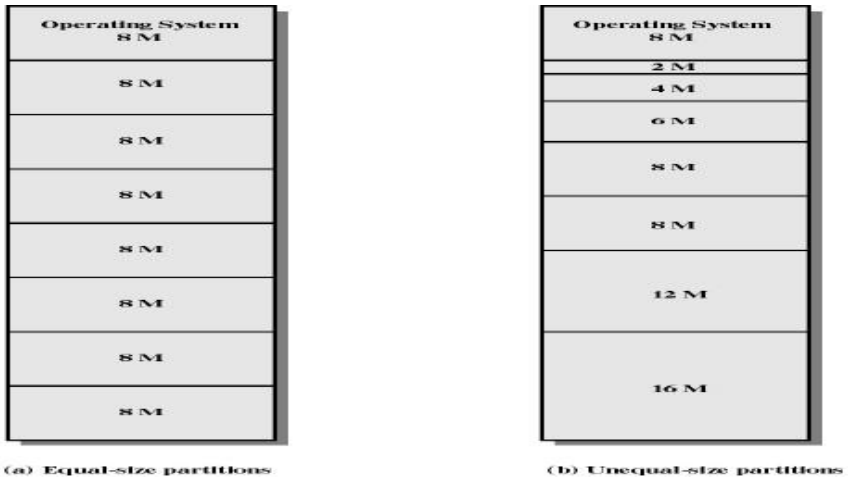


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Figure 65: Example of fixed partitioning of a 64-Mbyte memory

Equal-size partitions

- because all partitions are of equal size, it does not matter which partition is used

Unequal-size partitions

- can assign each process to the smallest partition within which it will fit
- queue for each partition
- processes are assigned in such a way as to minimize wasted memory within a partition

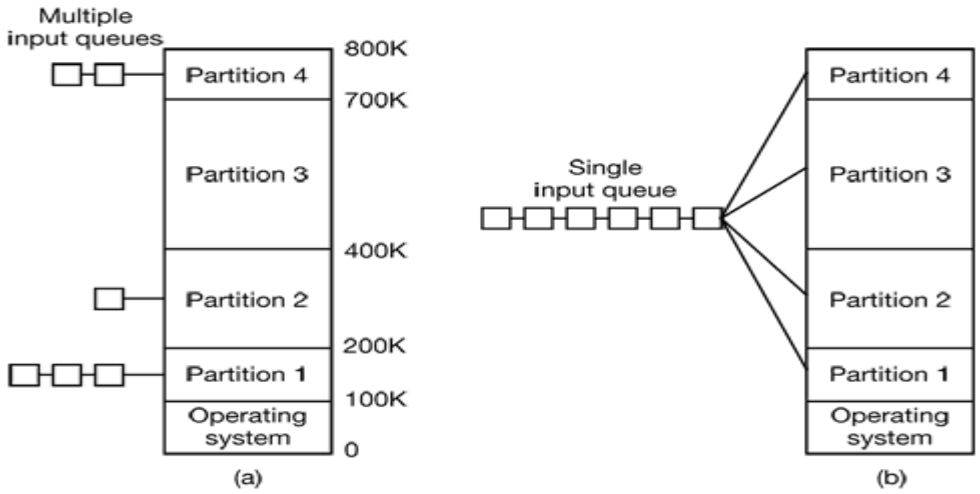


Figure 66: (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue

Advantages of fixed partition

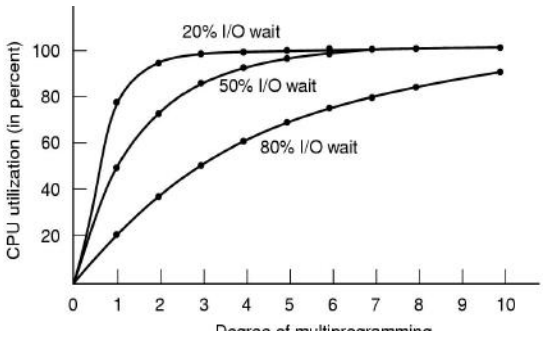
- Implementation of this allocation scheme is simple.
- The overhead of processing is also slow.
- It supports multiprogramming.
- It requires no special costly hardware.
- It makes efficient utilization of processor and I/O devices.

Disadvantages of fixed partitioning

- No single program (process) may exceed the size of the largest partition in a given system.
- It does not support a system having dynamic data structure such as stack, queue, heap etc.
- It limits the degree of multiprogramming which in turn may reduce the effectiveness of short-term scheduling.
- The Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This is called ***internal fragmentation***.

Modeling Multiprogramming (Probabilistic viewpoint of CPU usage)

- Let p=the fraction of time waiting for I/O to complete.
- n= no. of processes in the memory at once.
- The probability that all n processes are waiting for I/O(CPU idle time) =p^n
- So, CPU utilization = 1- p^n
- Following diagram shows the CPU utilization as a function of *n(degree of multiprogramming)*



- Fig: CPU utilization as a function of number of processes in memory

Example

Let, total memory = 1M =1000K

Memory space occupied by OS = 200K
Memory space taken by an user program = 200K
NOW

Number of processes n= (1000-200)/200 =4
[n= total user space/size of an user program]

CPU utilization = 1- $(0.8)^4$ =60%

- Add another 1M memory, then

n= (2000-200)/200 =9

CPU utilization = 1- $(0.8)^9$ =87%

Improvement = (87-60)/60 = 45%

- Again add another 1M, then

n= (3000-200)/200 =14

CPU utilization = 1- $(0.8)^{14}$ =96%

Improvement = (96-87)/87 = 10% improvement

Conclusion: addition of last 1M is not logical.

3.1.1.2. Multiprogramming with Variable Partitions

To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed. The partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more. An example, using 64 Mbytes of main memory, is shown in Figure

Eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as ***external fragmentation***, indicating that the memory that is

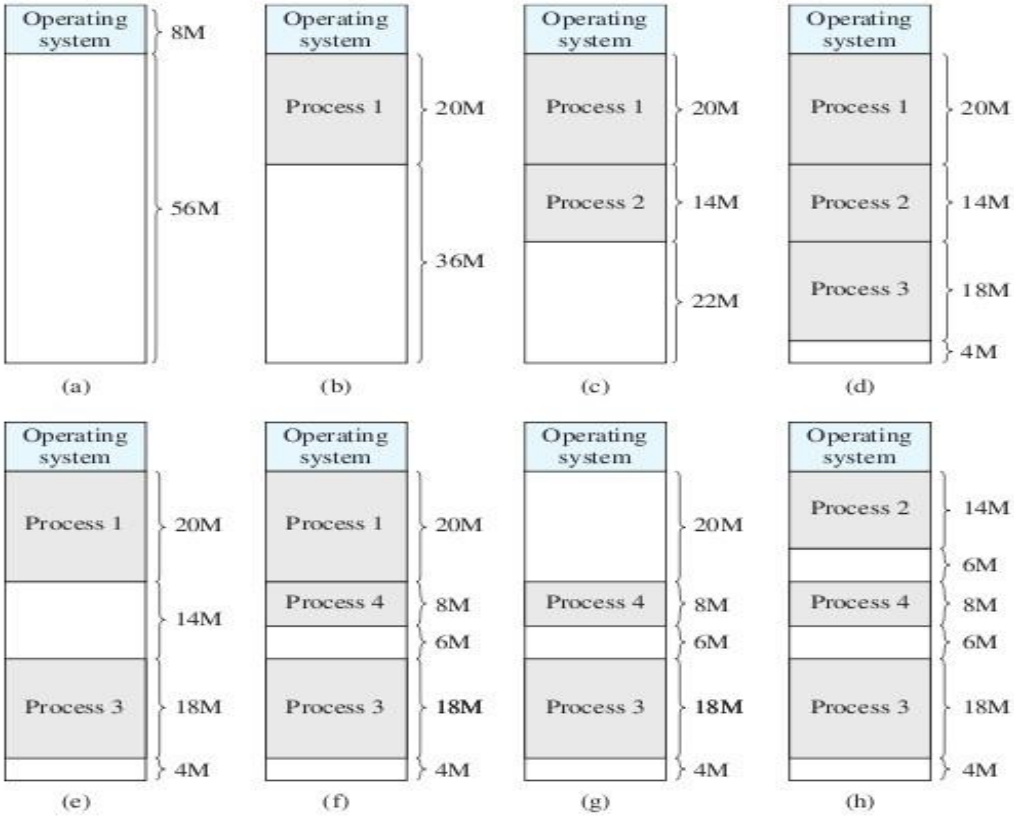
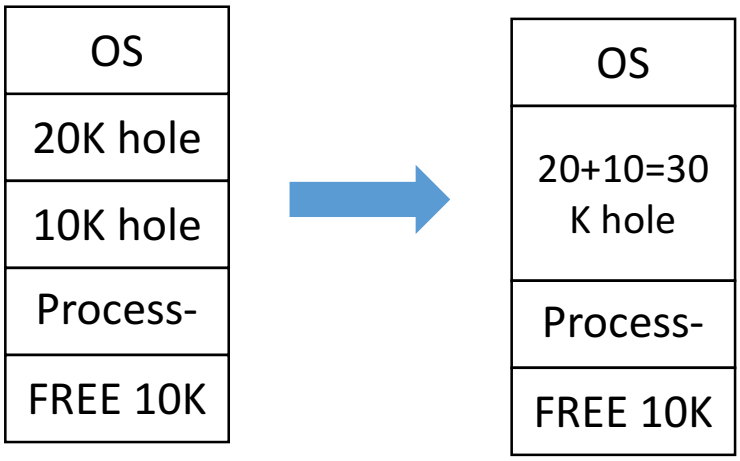


Figure 67: Effects of dynamic programming



external to all partitions becomes increasingly fragmented.
One technique for overcoming *external fragmentation* is *compaction*: From time to time, the operating system shifts the processes so that they are contiguous and so that all of the free memory is together in one block. For example, in Figure h, compaction will result in a block of free memory of length 16M.

Figure 68: Coalescing

This may well be sufficient to load in an additional process. The difficulty with compaction is that it is a time consuming procedure and wasteful of processor time.

- A process can grow if there is an adjacent hole.
- Otherwise the growing process is moved to the hole large enough for it or swap out one or more processes to disk.
- It has also some degree of waste.
- When a process finishes and leaves hole, the hole may not be large enough to place new job.
- Thus, variable partition multiprogramming, waste does occur.
- Following two activities should be taken place, to reduce wastage of memory:
  (a) Coalescing
  (b) Compaction

a) Coalescing

The process of merging two adjacent holes to form a single larger hole is called coalescing.

b) Compaction

- Even when holes are coalesced, no individual hole may be large enough to hold the job, although the sum of holes is larger than the storage required for a process.
- It is possible to combine all the holes into one big one by moving all the processes downward as far as possible; this technique is called memory compaction.

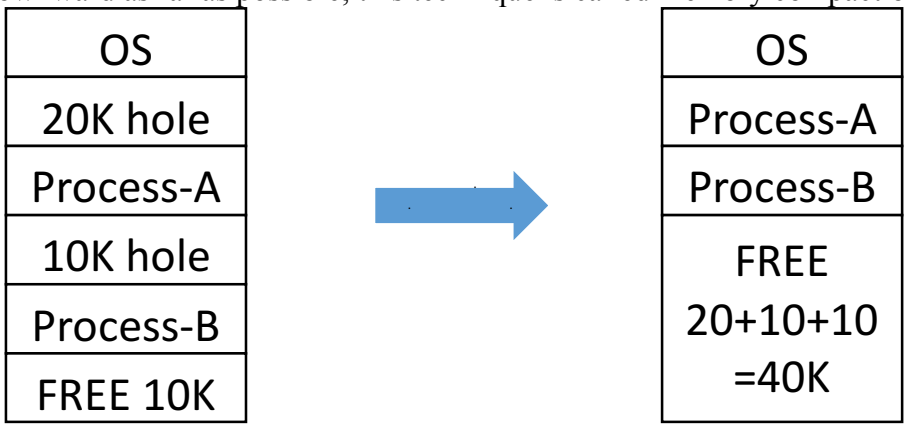


Figure 69: Compaction

Drawbacks of Compaction

- Reallocation info must be maintained.
- System must stop everything during compaction.
- Memory compaction requires lots of CPU time.
- For example:
    - On a 256MB machine that can copy 4 bytes in 40nses, it takes 2.7sec to compact all of memory.

3.4.1.3 Fixed vs. Variable Partitioning

Fixed Partition	Variable partition
• It is the OS that decides the partition size only once at the system boot time.	• OS has to decide about partition size, every time a new process is chosen by long-term scheduler.
• Here, the degree of multiprogramming is fixed.	• Here, the degree of multiprogramming will vary depending on program size.
• It leads to internal fragmentation.	• It leads to external fragmentation.
• IBM-360 DOS and OS/MFT OS used this approach.	• IBM OS/MFT used this approach also.

Table 7: Fixed partition vs Variable partition

Swapping

- If there is not enough main memory to hold all the currently active processes, the excess processes must be kept on the disk and brought in to run dynamically.
- Swapping consists of moving processes from main memory and disk.
- Relocation may be required during swapping.
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

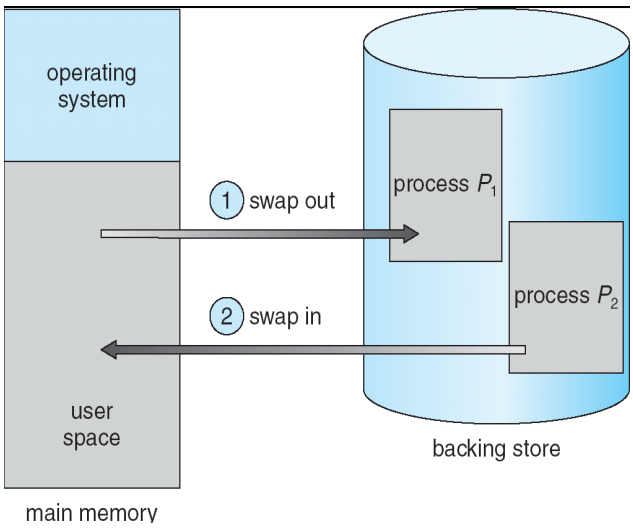


Figure 70: Swapping of two processes using a disk as a blocking store

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

A process, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. 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 in another process to the memory space that has been freed. When each process finishes its quantum, it will be swapped with another process.

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 so that it can load and execute the higher priority process. When the higher priority process finishes, the lower- priority process can be swapped back in and continued. This variant of swapping is sometimes called rollout, roll in. A process is swapped out will be swapped back into the same memory space that it occupies previously. If binding is done at assembly or load time, then the process cannot be moved to different location. If execution-time binding is being used, then it is possible to swap a process into a different memory space.

The context-switch time in such a swapping system is fairly high. Let us assume that the user process is of size 100K and the backing store is a standard hard disk with transfer rate of 1 megabyte per second. The actual transfer of the 100K process to or from memory takes

100K / 1000K per second = 1/10 second = 100 milliseconds

3.1.2. Storage placement Strategies (Dynamic Partitioning Placement Algorithm)

3.5.1.1. First fit:

- The memory manager allocates the first hole that is big enough. It stops the searching as soon as it finds a free hole that is large enough.
- **Advantages**: It is a fast algorithm because it searches as little as possible.
- **Disadvantages**: Not good in terms of storage utilization.

3.5.1.2.Next fit

- It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

3.5.1.3.Best fit:

- Allocate the smallest hole that is big enough.
- Best fit searches the entire list and takes the smallest hole that is big enough to hold the new process.
- Best fit try to find a hole that is close to the actual size needed.
- **Advantages**: more storage utilization than first fit.
- **Disadvantages**: slower than first fit because it requires searching whole list at time.

3.5.1.4.Worst fit:

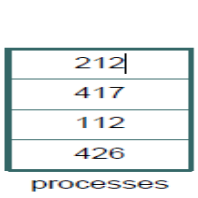
- Allocate the largest hole.
- It search the entire list, and takes the largest hole, rather than creating a tinny hole, it produces the largest leftover hole, which may be more useful.
- **Advantages**: some time it has more storage utilization than first fit and best fit.
- **Disadvantages**: not good for both performance and utilization.

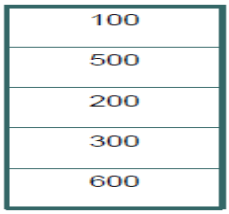
3.5.1.5.Quick Fit:

- Maintains separate lists for some of the more common sizes requested.
- For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on.
- Holes of say, 21 KB, could either be put on the 20-KB list or on a special list of odd-sized holes.
- With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive.
- If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

Q: Given the memory partitions of 100K, 500K, 200K, 300K and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K,112K, and 426K (in order)?

Which algorithm makes the most efficient use of memory?



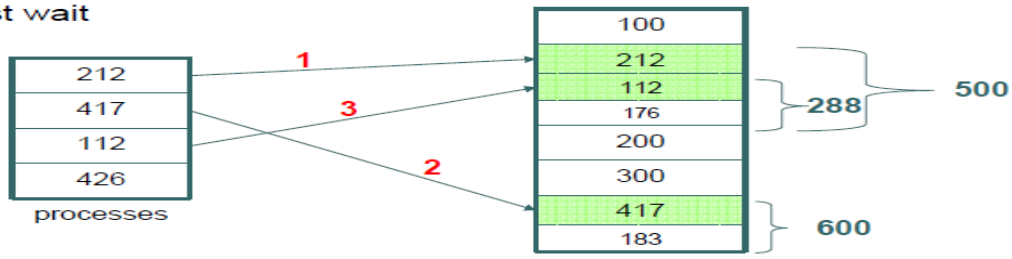


Example

- First-fit: search the list of available memory and allocate the first block that is big enough

Processes placement:

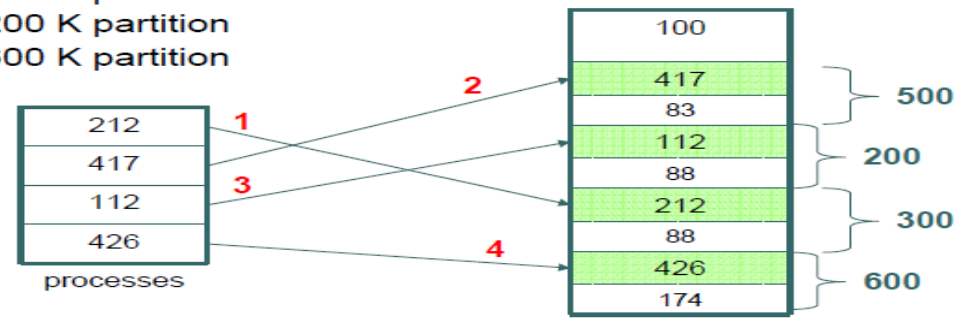
212 K → 500 K partition
417 K → 600 K partition
112 K → 288 K partition (New partition 288 K = 500 K-212 K)
426 K must wait



- Best-fit: search the entire list of available memory and allocate the smallest block that is big enough

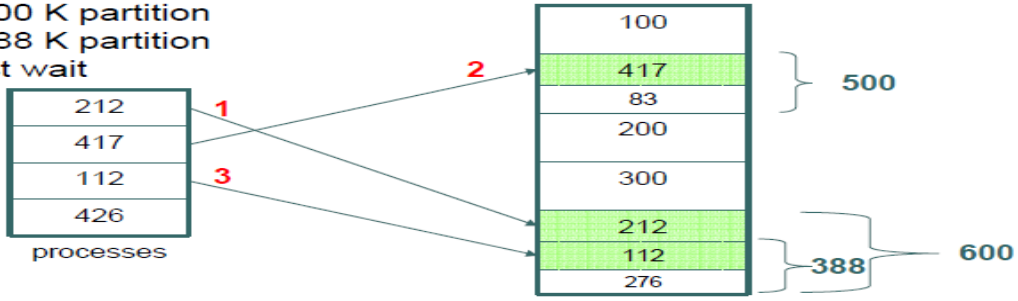
Processes placement:

212 K → 300 K partition
417 K → 500 K partition
112 K → 200 K partition
426 K → 600 K partition



- Worst-fit: search the entire list of available memory and allocate the largest block. The justification for this scheme is that the leftover block produced would be larger and potentially more useful than that produced by the best-fit approach

Processes placement:
212 K → 600 K partition
417 K → 500 K partition
112 K → 388 K partition
426 K must wait

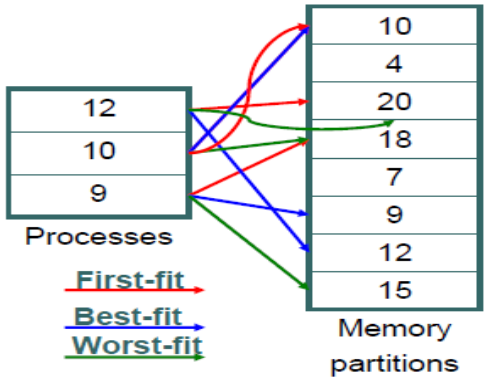


Note: In this example Best-fit turns out to be the best, since it allocates memory for all processes

Q: Consider a swapping system in which memory consists of the following hole sizes in memory order: 10K, 4K, 20K, 18K, 7K, 9K, 12K, and 15K. Which hole is taken for successive segment requests of:

I) 12K
II) 10K
III) 9K
for
(a) First-fit?
(b) Best-fit?
(c) Worst-fit?

	First-Fit	Best-Fit	Worst-Fit
12	20	12	20
10	10	10	18
9	18	9	15



3.5.2. Paging

External fragmentation is avoided by using paging. In this physical memory is broken into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory **frames**. Every address generated by the CPU is divided into any two parts: a *page number (p)* and a *page offset (d)* (Fig). The page number is used as an index into a *page table*. To run a program of size n pages, need to find n free frames and load program. Set up a page table to translate logical to physical addresses. The page table contains the base address of each page in physical memory. This base address is combined with the gage offset to define the physical memory address that is sent to the memory unit.

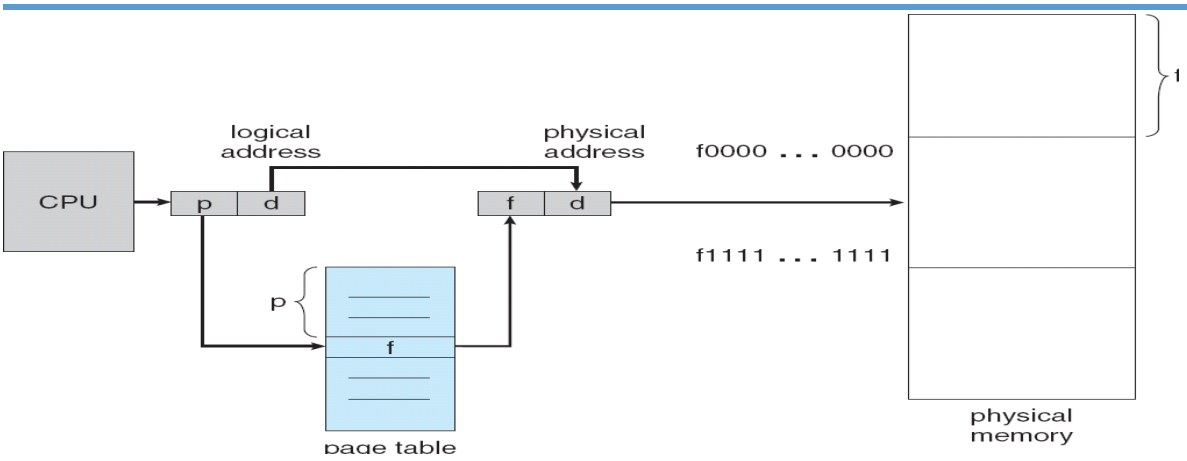
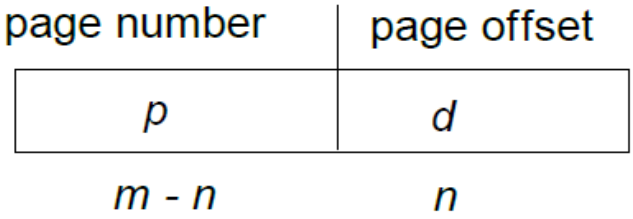


Figure 71: Paging Hardware

The page size like is defined by the hardware. The size of a page is typically a power of 2 varying between 512 bytes and 8192 bytes per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset. lf the size of logical address space is 2^m, and a page size is 2^n addressing units (bytes or words), then the high-order *m - n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset. Thus, the logical address is as follows:



Where *p* is an index into the page table and *d* is the displacement within the Paging is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address.

When we use a paging scheme, we have no external fragmentation: Any free frame can be allocated to a process that needs it.

If process size is independent of page size, we can have internal fragmentation to average one-half page per process.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, there must be at least n frames available in memory. If there are n frames available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames and the frame number is put in the page table for this process. The next page is loaded into another frame, and its frame number is put into the page table, and so on.

The user program views that memory as one single contiguous space, containing only this one program. But the user program is scattered throughout physical memory and logical addresses are translated into physical addresses.

The operating system is managing physical memory, it must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a

frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free allocated and, if it is allocated, to which page of which process or processes.

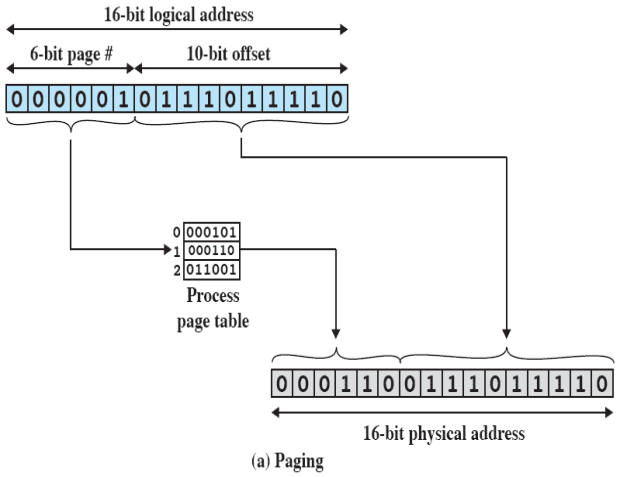
The operating system maintains a copy of the page table for each process. Paging therefore increases the context-switch time.

Example:

In this example, we have the logical address 0000010111011110, which is page number 1, offset 478.

Suppose that this page is residing in main memory frame 6 = binary 000110.

Then the physical address is frame number 6, offset 478 = 0001100111011110



3.5.2.1.Paging Model of Logical and Physical Memory

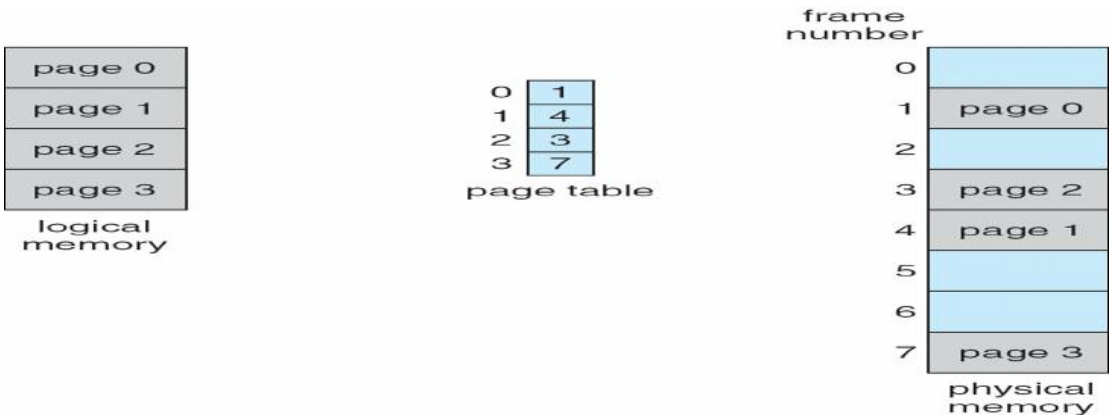
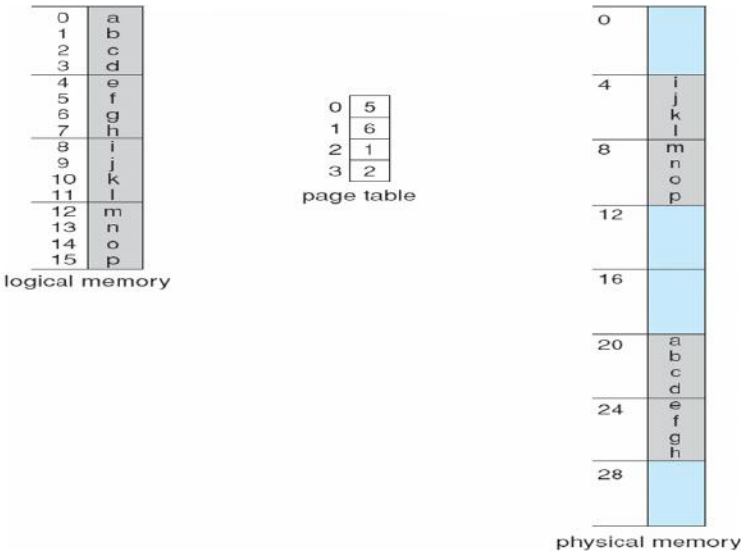


Figure 72: Paging Model



Paging Example

- Here, in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).
- Logical address 0 is page 0, offset 0.
- Indexing into the page table, we find that page 0 is in frame 5.
- Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0].

Figure 73: 32-byte memory and 4-byte pages

Free Frames

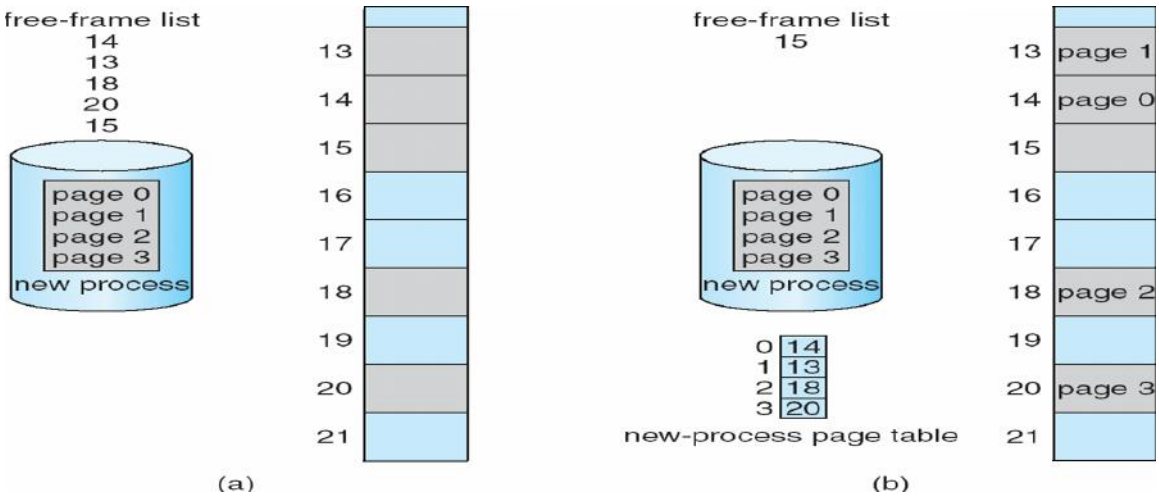
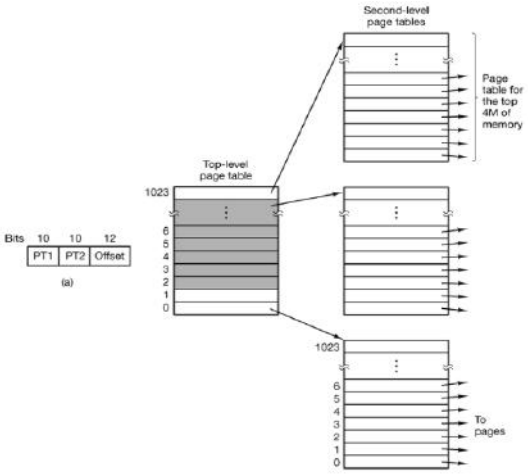


Figure 74: a) Before allocation b) After Allocation

3.5.2.2.Multi-level page tables



- Want to avoid keeping the entire page table in memory because it is too big
- Hierarchy of page tables
- The hierarchy is a page table of page tables

Figure 75: (a) A 32-bit address with two page table fields.

3.5.2.3.Implementation of Page Table

- Page table is kept in main memory.
- *Page-table base register (*PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffers (TLBs)*
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory

Page	Frame #

- Associative memory – parallel search
- Address translation (p, d)
- If p is in associative register, get frame out
- Otherwise get frame # from page table in memory

3.5.2.4.Paging Hardware with TLB

- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.
- The whole task may take less than 10 percent longer than it would if an unmapped memory reference were used.

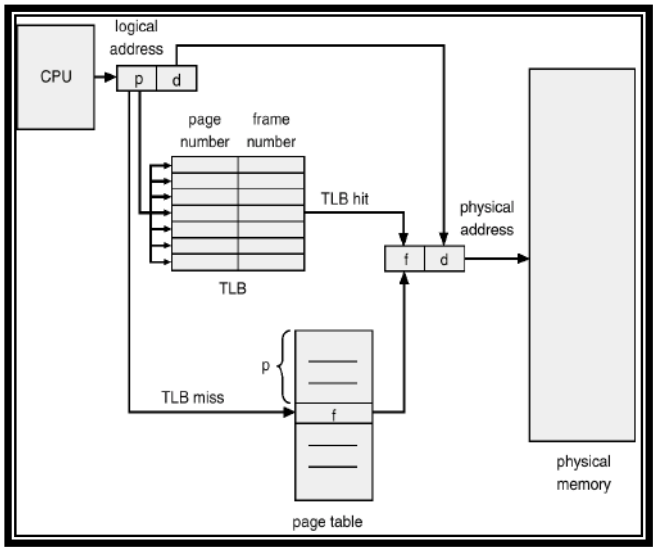
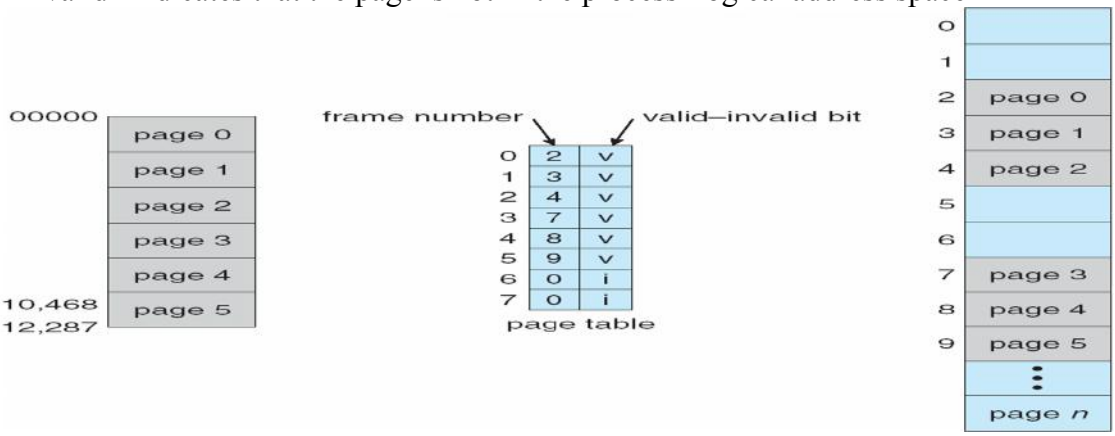


Figure 76: Paging Hardware with TLB

- If the page number is not in the TLB (known as TLB miss), a memory reference to the page table must be made.
- When the frame number is obtained, we can use it to access memory.
- In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

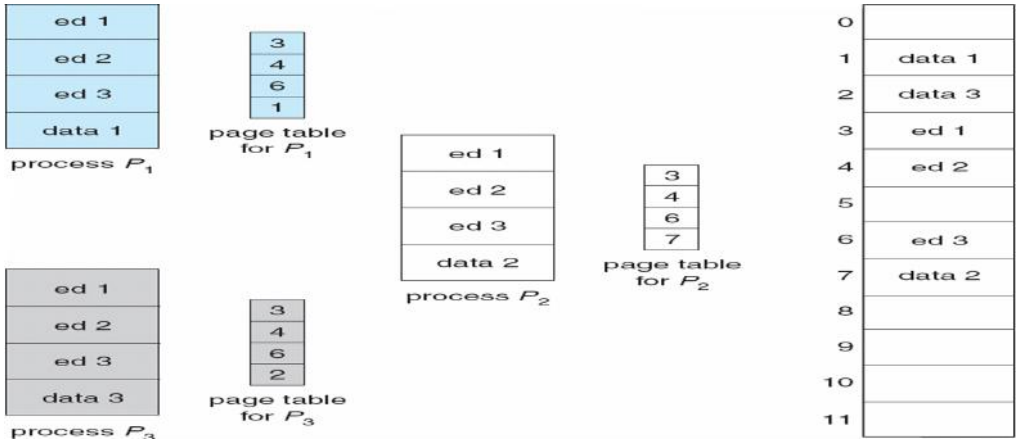
3.5.2.5.Memory Protection

- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
- "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
- "invalid" indicates that the page is not in the process' logical address space



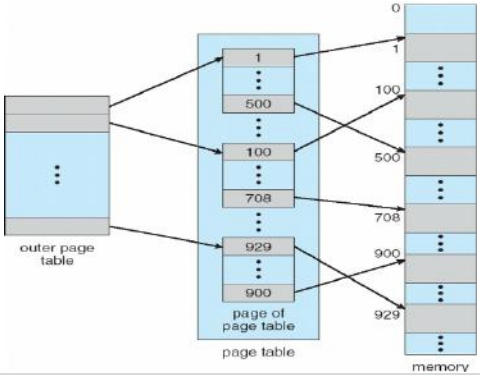
3.5.2.6.Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- Private code and data
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space.



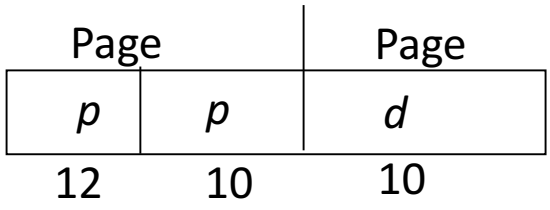
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

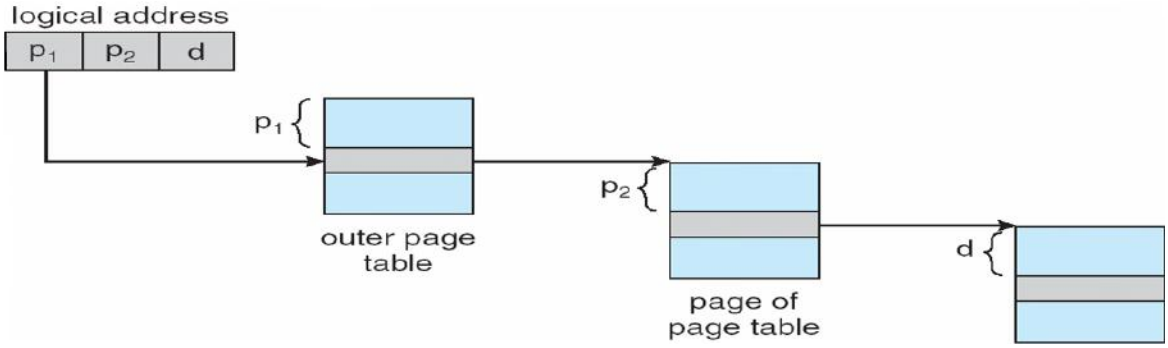


3.5.2.7.Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- A logical address (on 32-bit machine with 1K page size) is divided into:
- a page number consisting of 22 bits
- a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
- a 12-bit page number
- a 10-bit page offset
- Thus, a logical address is as follows:



where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

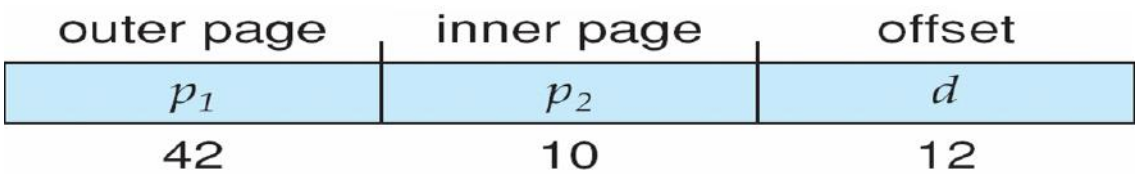


Address-Translation Scheme

Fig: Address translation scheme

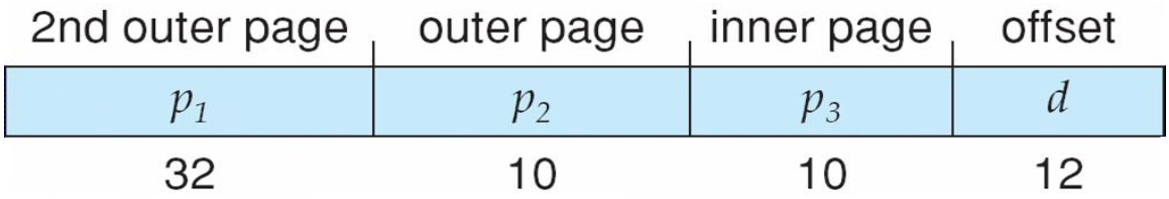
Three-level Paging Scheme

- For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate.
- Let us suppose that the page size in such a system is 4 KB.
- In this case, the page table consists of up to 2^{52} entries.
- If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries.



- The outer page table consists of 2^{42} entries.
- The obvious way to avoid such a large table is to divide the outer page table into smaller pieces.
- Suppose that the outer page table is made up of standard-size pages (2^{10} entries, or 2^{12} bytes).

- In this case, a 64-bit address space is looks as follows:



3.5.2.8.Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
- This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
- If a match is found, the corresponding physical frame is extracted

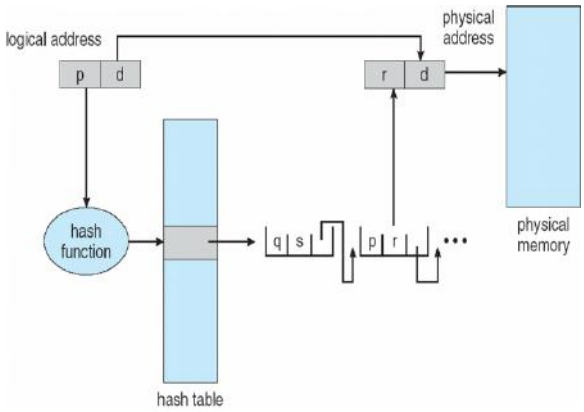


Figure 77: Hashed Page Tables

3.5.2.9.Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

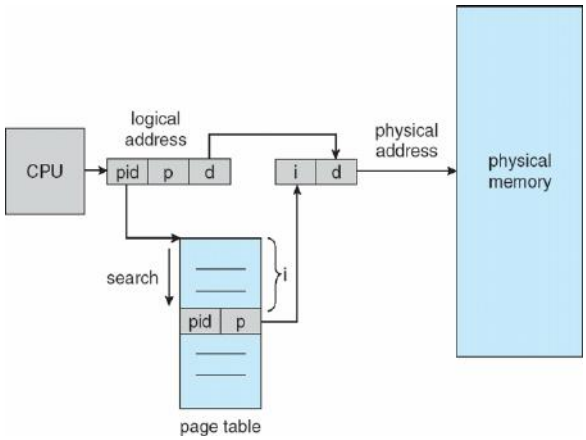


Figure 78: Inverted Page Table

3.5.3. Segmentation

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of *segments*. Segmentation is the Memory-management scheme that supports user view of memory. A program is a collection of segments. A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table, arrays.

It is not required that all segments of all programs be of the same length, although there is a maximum segment length. As with paging, a logical address using segmentation consists of two parts, in this case a *segment number* and an *offset*.

Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning. In segmentation, a program may occupy more than one partition, and these partitions need

not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less. Whereas paging is invisible to the programmer, segmentation usually visible and is provided as a convenience for organizing programs and data.

Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses. Segmentation scheme would make use of a segment table for each process and a list of free blocks of main memory. Each segment table entry would have to as in paging give the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment, to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware.

Consider an address of n + m bits, where the leftmost n bits are the segment number and the rightmost m bits are the offset. The following steps are needed for address translation:

- Extract the segment number as the leftmost n bits of the logical address.
- Use the segment number as an index into the process segment table to find the starting physical address of the segment.
- Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
- The desired physical address is the sum of the starting physical address of the segment plus the offset.
- Segmentation and paging can be combined to have a good result.

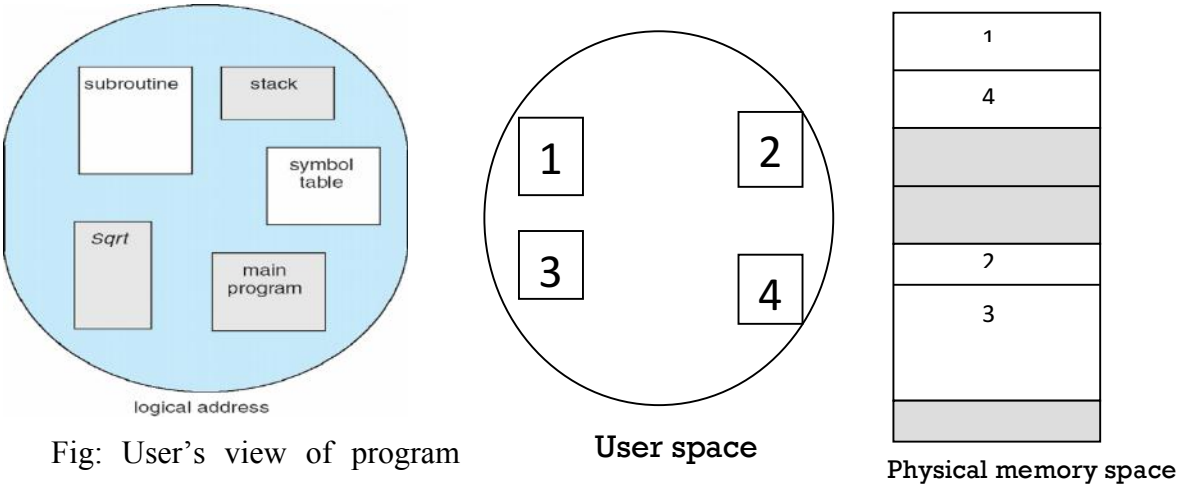


Fig: User's view of program

Figure 79: Logical view of segmentation

3.5.3.1.Segmentation Architecture

- Logical address consists of a two tuple:
  - <segment-number, offset>,
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
- base – contains the starting physical address where the segments reside in memory.
- *limit* – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory.

- *Segment-table length register (STLR)* indicates number of segments used by a program;
- segment number *s* is legal if *s* < STLR.

Protection

- With each entry in segment table associate:
- validation bit = 0 ⇒ illegal segment
- read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

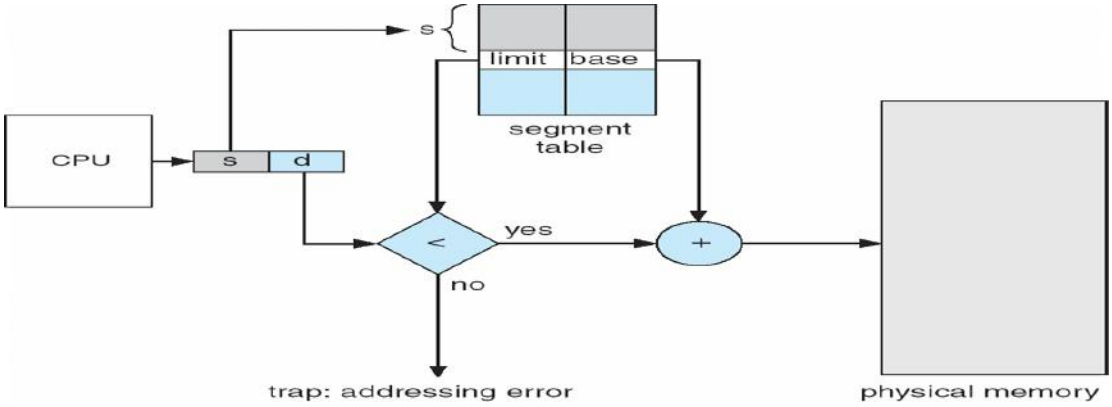


Figure 80: Segmentation hardware

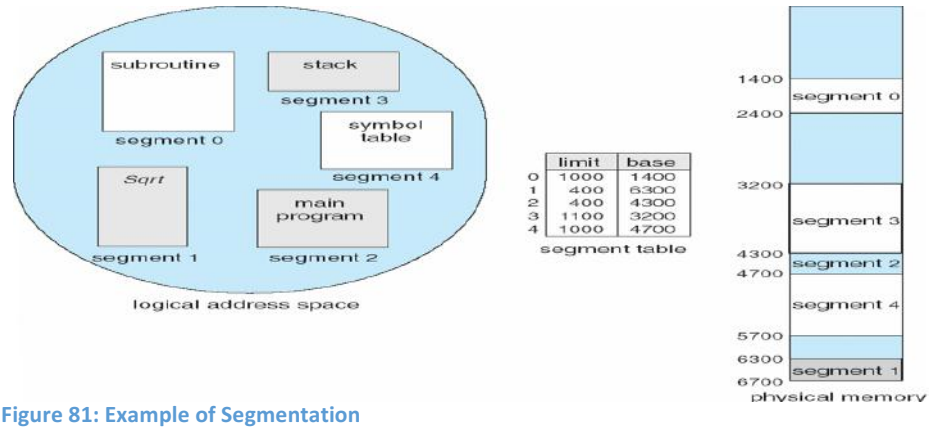


Figure 81: Example of Segmentation

3.5.3.2.Paging vs Segmentation

SNo	Paging	Segmentation
1	Block replacement easy Fixed-length blocks	Block replacement hard Variable-length blocks Need to find contiguous, variable-sized, unused part of main memory
2	Invisible to application programmer	Visible to application programmer.

3	No external fragmentation, But there is Internal Fragmentation unused portion of page.	No Internal Fragmentation, But there is external Fragmentation unused portion of main
4	Units of code and date are broken into separate pages.	Keeps blocks of code or data as a single units.
5	Segmentation is a logical unit visible to the user's program and id of arbitrary	Paging is a physical unit invisible to the user's view and is of fixed size
6	Segmentation maintains multiple address spaces per process.	Paging maintains one address space.
7	No sharing of procedures between users is facilitated.	Sharing of procedures between users is facilitated.
8	OS must maintain a free frame list	OS maintain the list of free holes in memory

Table 8: Paging vs Segmentation

3.5.3.3.Segmentation with paging

- The logical address space of a process is divided into two partition
- The first partition consists of up to 8k segments that are private to that process
- The second partition consists of up to 8k segments that are shared among all the processes.
- Information about first partition is kept in the local descriptor table (LDT).
- Information about second partition is kept in the global descriptor table (GDT).
- Each entry in the LDT and GDT table consists of 8 bytes with detailed information about a particular segment including the base location and length of that segment.
- The logical address is a pair (selector, offset)
- The selector is a 16-bit number:

  - Where ***s*** designates the segment number,
  - ***g*** indicates whether the segment is in GDT or LDT and
  - ***p*** deals with protection.
- The offset is a 32-bit number specifying the location of the byte within the segment

Virtual Memory

- Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory.
- Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (**Fig**).

Following are the situations, when entire program is not required to load fully.

1. User written error handling routines are used only when an error occurs in the data or computation.
1. Certain options and features of a program may be used rarely.
2. Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.

The ability to execute a program that is only partially in memory would counter many benefits.

1. Less number of I/O would be needed to load or swap each user program into memory.
2. A program would no longer be constrained by the amount of physical memory that is available.
3. Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

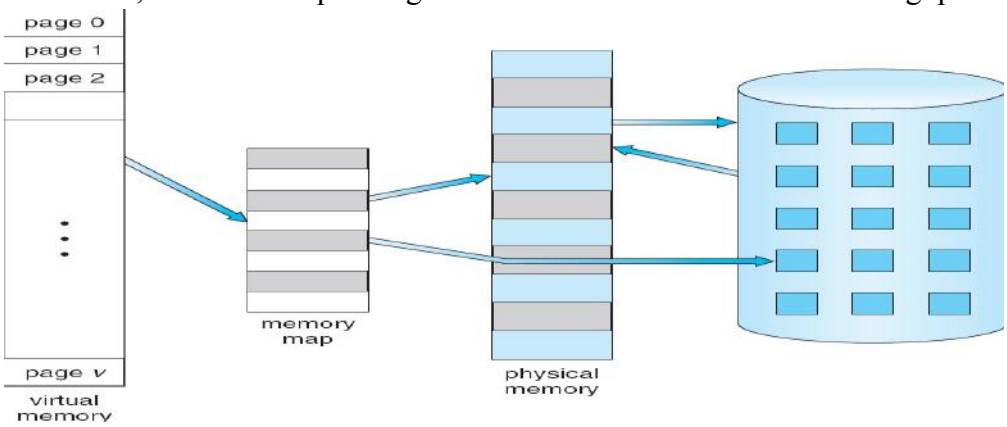


Figure 82: Diagram showing virtual memory that is larger than physical memory.

Virtual memory is commonly implemented

- Demand paging.
- Demand segmentation

3.6.1. Demand Paging

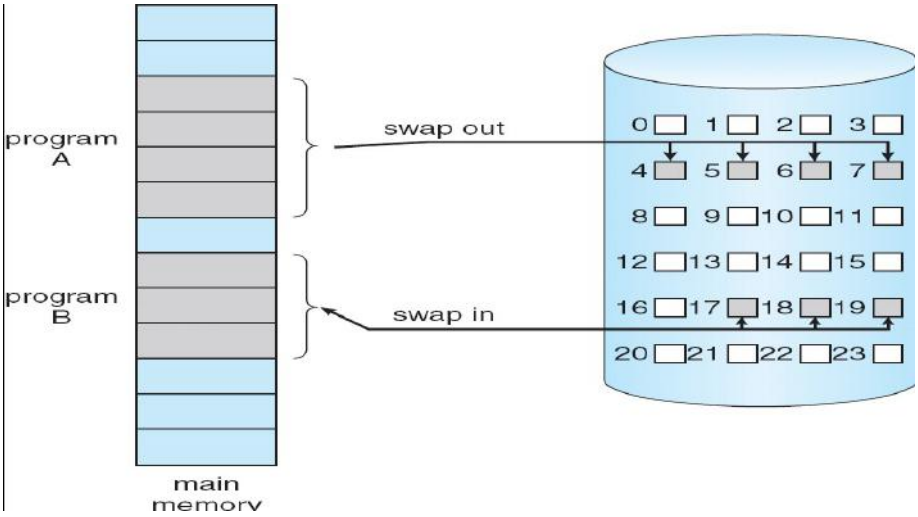


Figure 83: Transfer of a paged memory to continuous disk space

A demand paging is similar to a paging system with swapping (Figure 83). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory. This is done by lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are viewing process as sequence of pages instead of swapper we use the term pager.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not

be used in anyway, decreasing the swap time and the amount of physical memory needed.

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme. Where valid and invalid pages can be checked checking the bit and marking a page will have no effect if the process never attempts to access the pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

Access to a page marked invalid causes a page-fault trap. This trap is the result of the operating system's failure to bring the desired page into memory. But page fault can be handled as following (Fig 8.3):

- Bring a page into memory only when it is needed.
    - Less I/O needed
    - Less memory needed
    - Faster response
    - More users
- Page is needed ⇒ reference to it
- invalid reference ⇒ abort
- not-in-memory ⇒ bring to memory

Some of the pages belonging to this process are in memory, and some are on the disk.

A bit in the page table tells where to find the page.

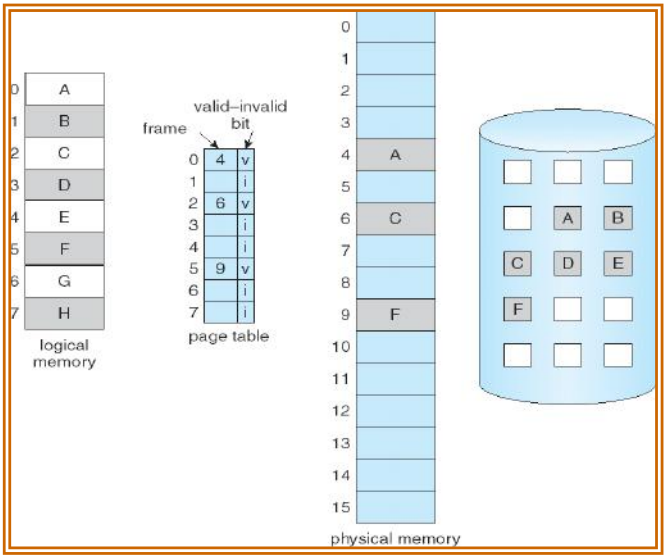


Figure 84: The Picture When All Pages Are Not In Memory

Page fault/when does page fault happens???

- We know when process is executed it needs pages.
- If the process tries to use a page that was not brought into memory, this situation is called page fault trap.
- When a page fault happens we need to handle it.

1. We check an internal table for this process to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If .it was valid, but we have not yet brought in that page, we now page in the latter.
1. We find a free frame.

2. We schedule a disk operation to read the desired page into the newly allocated frame.
3. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
4. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been memory.

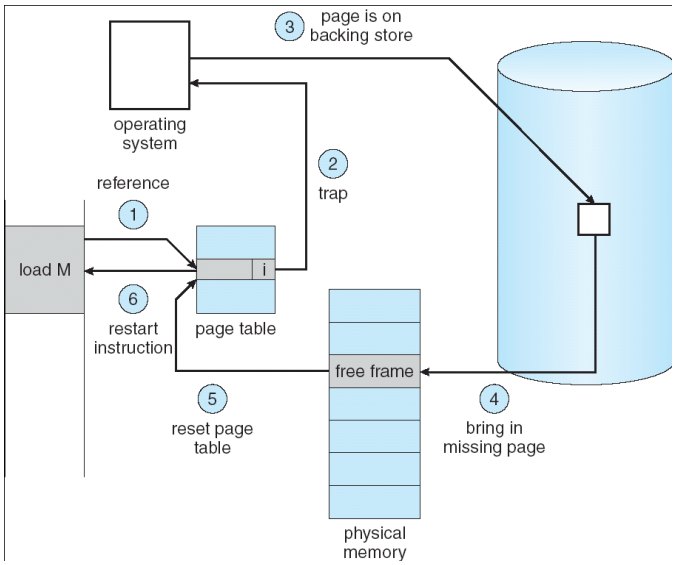


Figure 85: The Picture When All Pages Are Not In Memory

Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

The page replacement is used to make the frame free if they are not in used. If no frame is free then other process is called in.

3.6.1.1. Advantages of Demand Paging:

1. Large virtual memory.
2. More efficient use of memory.
3. Unconstrained multiprogramming. There is no limit on degree of multiprogramming.

3.6.1.2.Disadvantages of Demand Paging:

1. Number of tables and amount of processor over head for handling page interrupts are greater than in the case of the simple paged management techniques.
2. Due to the lack of an explicit constraints on a jobs address space size.

3.6.2. Page Replacement

- When we over-allocate memory, we need to push out something already in memory. Over-allocation may occur when programs need to fault in more pages than there are physical frames to handle.

Approach:

- If no physical frame is free, find one not currently being touched and free it. Steps to follow are:
  - Find the location of the desired page on disk.
  - Find a free frame:

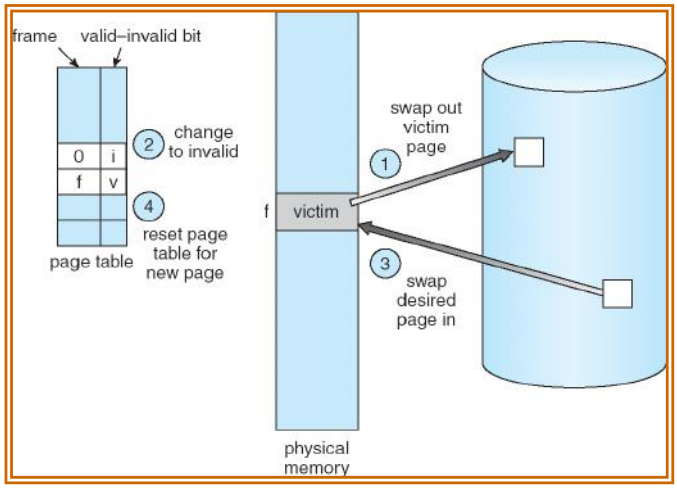


Figure 86: Page replacement

- - If there is a free frame, use it.
  - If there is no free frame, use a page replacement algorithm to select a *victim* frame.
- Read the desired page into the (newly) free frame. Update the page and frame tables.
- Restart the process.

3.6.3. Page Replacement Algorithm

There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults. The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data.

1. For a given page size we need to consider only the page number, not the entire address.
2. If we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

Some of the page replacement algorithms are

1. FIFO page replacement
2. Optimal page replacement
3. LRU page replacement
4. Second chance page replacement algorithm
5. Clock page replacement algorithm
6. Second chance page replacement algorithm

3.6.3.1.First In First out Page replacement algorithm

- The oldest and simplest page in the physical memory is the one selected for replacement.
- Very simple to implement.
- Keep a list on a page fault, the page at the head is removed and the new page added to the tail of the list.
- This algorithm associates with each page the time that page was brought into memory
- When a page must be replaced, the oldest page is chosen

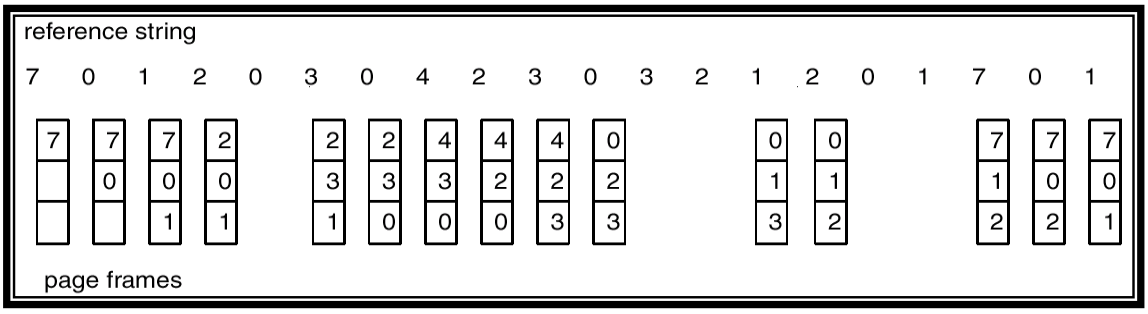


Figure 87: FIFO page replacement algorithm

Total numbers of page fault = 15

Fault rate = number of page fault/numbers of reference string
= 15/20 = ¾ = 0.75

To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available. As the number of frames available increase, the number of page faults will decrease.

3.6.3.2.Optimal Page Replacement

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN. It is simply replace the page that will not be used for the longest period of time i.e. future knowledge of reference string is required.

- Often called Balady's
- Min Basic idea: -Impossible to implement because it requires future knowledge of the reference string.

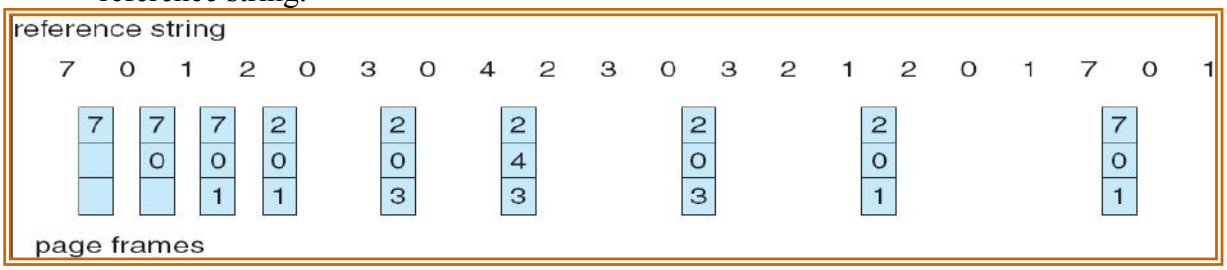


Figure 88: Optimal page replacement algorithm

Total numbers of page fault = 9
Fault rate = number of page fault/numbers of reference string
= 9/20 = 0.45

3.6.3.3.LRU Algorithm

The FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. In LRU replace the page that has not been used for the longest period of time.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.

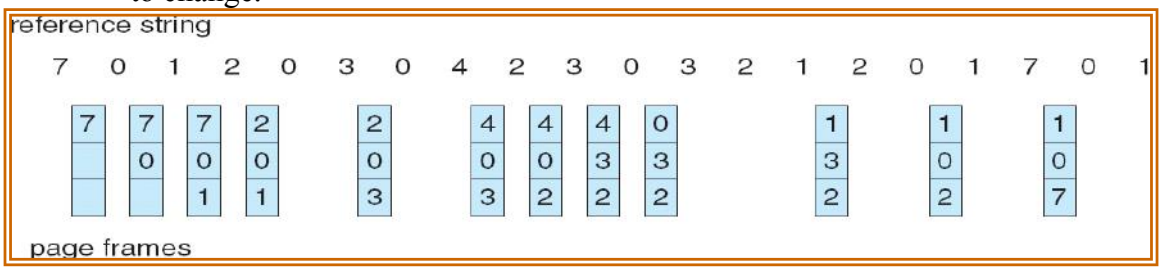


Figure 89: LRU page replacement algorithm

Total numbers of page fault = 12
Fault rate = number of page fault/numbers of reference string
= 12/20 = 0.6

3.6.3.4.LRU Approximation Algorithms

Some systems provide no hardware support, and other page-replacement algorithm. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced. Reference bits are associated with each entry in the page table initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

3.6.3.5.Additional-Reference-Bits Algorithm

The operating system shifts the reference bit for each page into the high- order or of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit.

These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

3.6.3.6.Second-Chance Algorithm

A simple modification to FIFO that avoids the problem of heavily used page. It inspects the R bit If it is 0, the page is both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

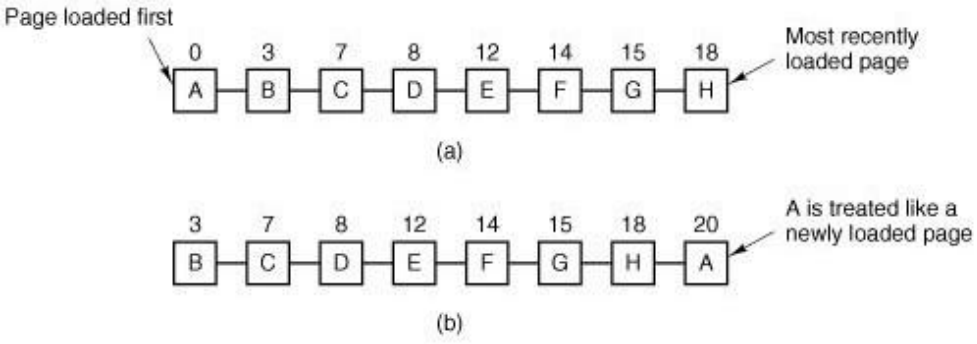


Fig: Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their loading times.

Suppose that a page fault occurs at time 20. The oldest page is *A,* which arrived at time 0, when the process started. If *A* has the *R* bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is dean). On the other hand, if the *R* bit is set, *A* is put onto the end of the list and its "load time" is reset to the current time (20). The *R* bit is also cleared. The search for a suitable page continues with *B.*

What second chance is doing is looking for an old page that has not been referenced in the previous clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO. Specifically, imagine that all the pages in above fig have their *R* bits set. One by one, the operating system moves the pages to the end of the list, clearing the *R* bit each time it

appends a page to the end of the list. Eventually, it comes back to page *A*, which now has its *R* bit cleared. At this point *A* is evicted. Thus the algorithm always terminates.

3.6.3.7.Counting Algorithms

There are many other algorithms that can be used for page replacement.

LFU Algorithm: The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

MFU Algorithm: The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

3.6.3.8.Page Buffering Algorithm

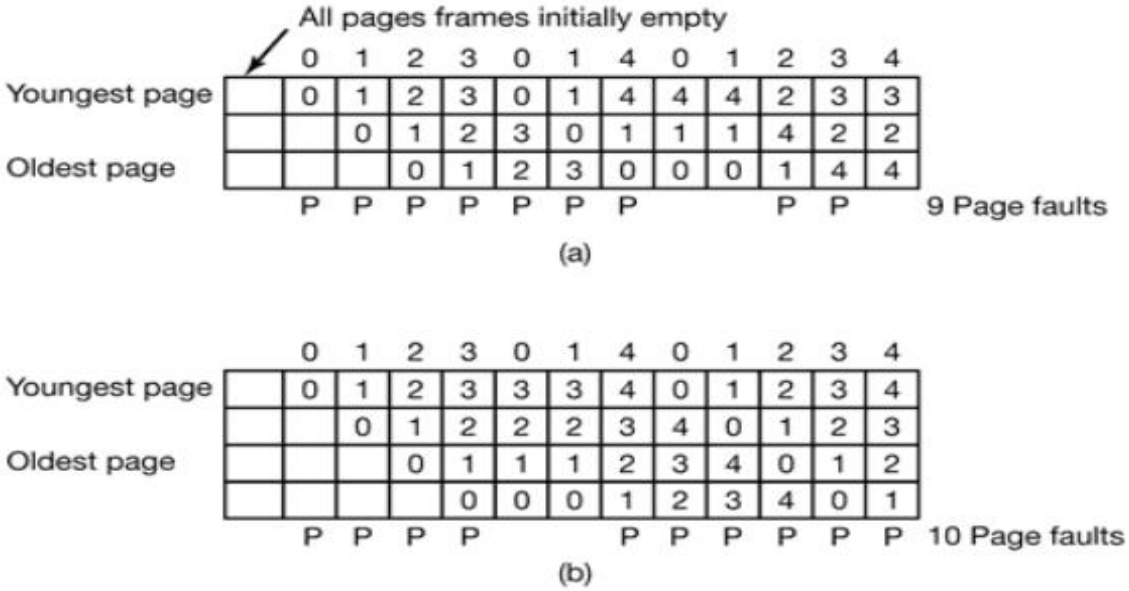
When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out.

This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

When the FIFO replacement algorithm mistakenly replaces a page mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm.

3.6.4. Belady's Anomaly

Intuitively, it might seem that the more page frames the memory has, the fewer page faults a program will get. Surprisingly enough, this is not always the case. Belady et al. discovered a counter example, in which FIFO caused more page faults with four page frames than with three. This strange situation has become known as **Belady's anomaly**. It is illustrated in Fig. for a program with five virtual pages, numbered from 0 to 4. The pages are referenced in the order



Reference String: 0 1 2 3 0 1 4 0 1 2 3 4

Figure 90: Belady's anomaly. (a) FIFO with three page frames. (b) FIFO with four page frames. The P's show which page references cause page faults.

3.6.5. Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high.This leads to:
  - Low CPU utilization.
  - Operating system thinks that it needs to increase the degree of multiprogramming.
  - Another process added to the system.
- **Thrashing** ≡ a process is busy swapping pages in and out.
- Why does paging work?
  - Locality model
- Process migrates from one locality to another.
- Localities may overlap.
- Why does thrashing occur?
- Σ size of locality > total memory size

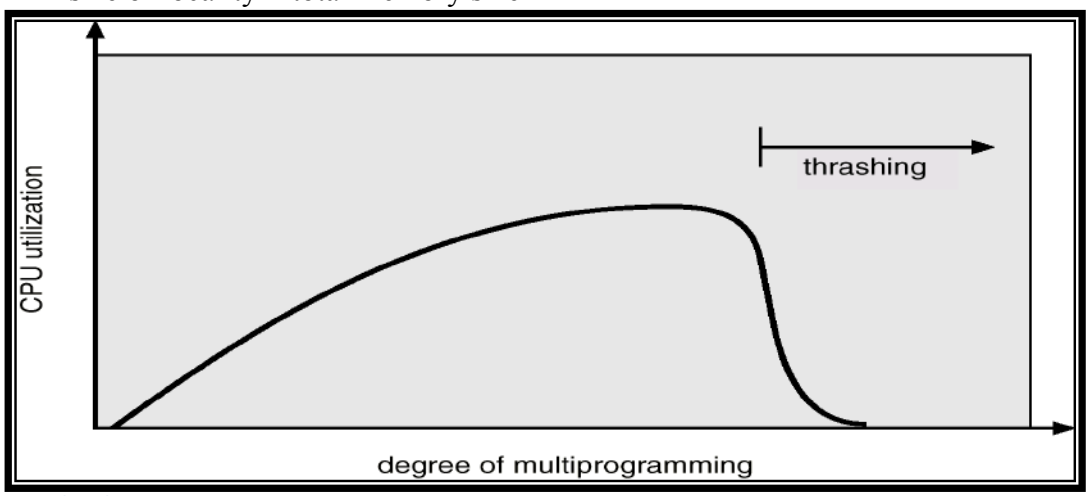


Figure 91: Thrashing

Summary

- Memory-management algorithms for multi programmed operating systems range from the simple single-user system approach to paged segmentation.
- 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.
  1. 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.
  2. Performance. As the memory-management algorithm becomes more complex, the time required to map a logical address to a physical address increases. 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.
  3. 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.
  4. 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.
  5. 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.
  6. 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.
  7. 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.
- It is desirable to be able to execute a process whose logical address space is larger than the available physical address space.
- Virtual memory is a technique that enables us to map a large logical address space onto a smaller physical memory. Virtual memory allows us to run extremely large

processes and to raise the degree of multiprogramming, increasing CPU utilization. It frees application programmers from worrying about memory availability. In addition, with virtual memory, several processes can share system libraries and memory. Virtual memory also enables us to use an efficient type of process creation known as copy-on-write, where in parent and child processes share actual pages of memory. Virtual memory is commonly implemented by demand paging.

- Pure demand paging never brings in a page until that page is referenced. The first reference causes a page fault to the operating system. The operating-system kernel consults an internal table to determine where the page is located on the backing store. It then finds a free frame and reads the page in from the backing store. The page table is updated to reflect this change, and the instruction that caused the page fault is restarted. This approach allows a process to run even though its entire memory image is not in main memory at once.
- As long as the page-fault rate is reasonably low, performance is acceptable. We can use demand paging to reduce the number of frames allocated to a process. This arrangement can increase the degree of multiprogramming (allowing more processes to be available for execution at one time) and-in theory, at least-the CPU utilization of the system.
- It also allows processes to be run even though their memory requirements exceed the total available physical memory. Such processes run in virtual memory. If total memory requirements exceed the capacity of physical memory, then it may be necessary to replace pages from memory to free frames for new pages.
- Various page-replacement algorithms are used. FIFO page replacement is easy to program but suffers from Belady's anomaly. Optimal page replacement requires future knowledge. LRU replacement is an approximation of optimal page replacement, but even it may be difficult to implement.
- To requiring that we solve the major problems of page replacement and frame allocation, the proper design of a paging system requires that we consider prepaging, page size, TLB reach, inverted page tables, program structure, I/0 interlock, and other issues.

CHAPTER 4

I/O MANAGEMENT

4.1. 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.

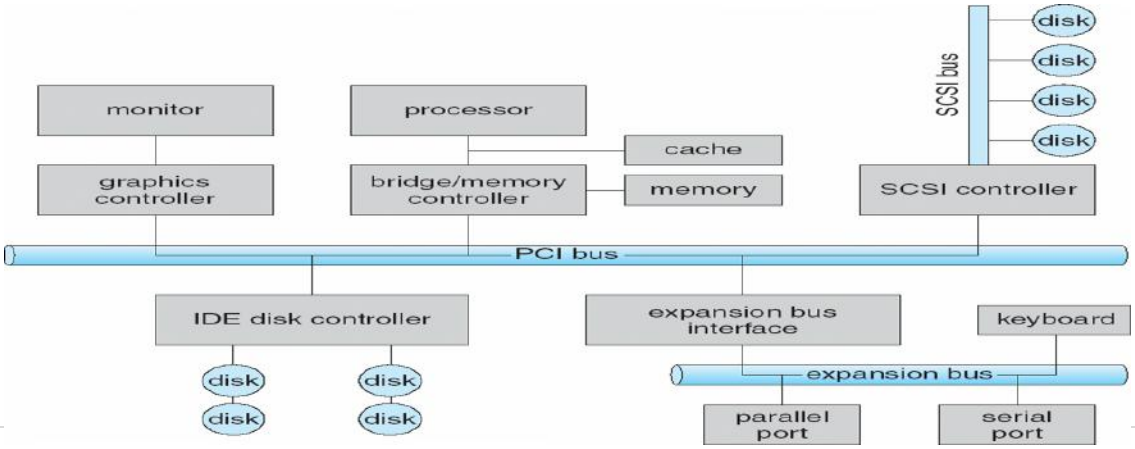


Figure 92: A typical PC bus structure

How can the processor give commands and data to a controller to accomplish an I/O transfer?

- Direct I/O instructions and Memory-mapped I/O

4.1.1. Direct I/O instructions

- The controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register.

4.1.2. Memory-mapped I/O

- The device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers. Especially for large address spaces (graphics)

4.2. 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 mean 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.
  1. The host repeatedly reads the busy bit until that bit becomes clear.
  2. The host sets the write bit in the command register and writes a byte into the data-out register.
  3. The host sets the command-ready bit.
  4. When the controller notices that the command-ready bit is set, it sets the Busy.
  5. The controller reads the command register and sees the write command.
  6. It reads the data-out register to get the byte, and does the I/O to the device.
  7. 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

4.3. Interrupts

The basic interrupt mechanism works as follows.

- 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 performs a state save 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, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt.
- **In Short**: The device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches it to the interrupt handler, and the handler clears the interrupt by servicing the device.

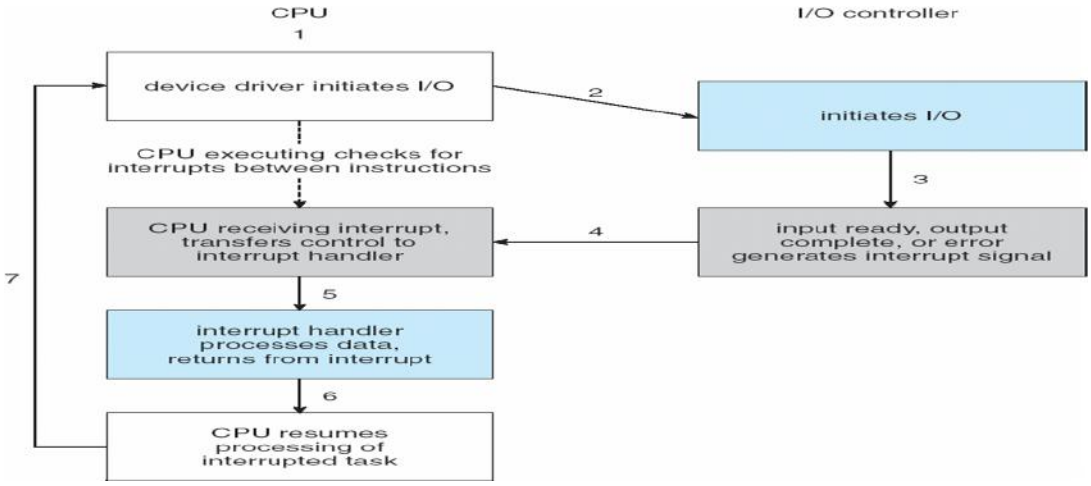


Figure 93: Interrupts driven I/O Cycle

- In a modern OS, we need more sophisticated interrupt handling features.
    1. We need the ability to defer interrupt handling during critical processing.
    2. 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.
    3. 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.
- Most CPUs have two interrupt request lines.
- One is the *non-maskable*, 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.
- The interrupt mechanism accepts an address, this address is an offset in a table called the Interrupt Vector Table (IVT).
- This vector contains the memory addresses of specialized interrupt handlers.
- The interrupt mechanism also implements a system of Interrupt Priority Level.
- 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.

- A modern OS interacts with the interrupt mechanism in several ways.
    - *At boot time*: the OS probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector.
    - *During I/O*: the various device controllers raise interrupts when they are ready for service.
- These interrupts signify that output has completed, or that input data are available, or that a failure has been detected.
- Interrupt mechanism also used for **exceptions.**
- Terminate process, crash system due to hardware error, accessing a protected memory address or attempting to execute privilege instruction from user mode, divide by zero etc.
- Page fault executes when memory access error.
- System call executes via **trap** to trigger kernel to execute request.
- Multi-CPU systems can process interrupts concurrently.
- If operating system designed to handle it.
- Used for time-sensitive processing, frequent, must be fast.

4.4. I/O Devices

Categories of I/O Devices

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

4.5. Ways to INPUT/OUTPUT

- There are three fundamentally different ways to do I/O
    1. Programmed I/O
    2. Interrupt-driven
    3. Direct Memory access

4.5.1. Programmed I/O

The processor issues an I/O command, on behalf of a process, to an I/O module; that process then busy waits for the operation to be completed before proceeding. When the processor is executing a program and encounters an instruction relating to input/output, it executes that instruction by issuing a command to the appropriate input/output module. With the programmed input/output, the input/output module will perform the required action and then set the appropriate bits in the input/output status register. The input/output module takes no further action to alert the processor. In particular it doesn't interrupt the processor. Thus, it is the responsibility of the processor to check the status of the input/output module periodically, until it finds that the operation is complete.
It is simplest to illustrate programmed I/O by means of an example. Consider a process that wants to print the eight character string ABCDEFGH.

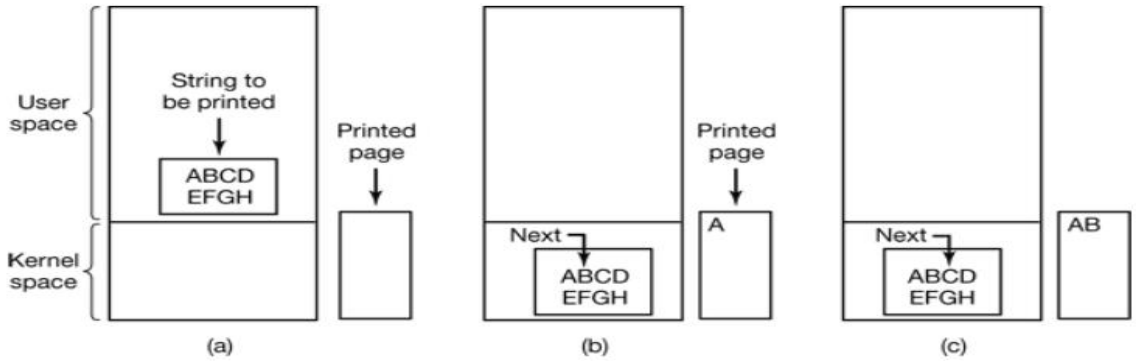


Figure 94: Steps in printing a string

1. It first assemble the string in a buffer in user space as shown in fig.
2. The user process then acquires the printer for writing by making system call to open it.
3. If printer is in use by other the call will fail and enter an error code or will block until printer is available, depending on OS and the parameters of the call.
4. Once it has printer the user process makes a system call to print it.
5. OS then usually copies the buffer with the string to an array, say P in the kernel space where it is more easily accessed since the kernel may have to change the memory map to get to user space.
6. As the printer is available the OS copies the first character to the printer data register, in this example using memory mapped I/O. This action activates the printer. The character may not appear yet because some printers buffer a line or a page before printing.
7. As soon as it has copied the first character to the printer the OS checks to see if the printer is ready to accept another one.
8. Generally printer has a second register which gives its status

The action followed by the OS are summarized in fig below. First data are copied to the kernel, then the OS enters a tight loop outputting the characters one at a time. The essentials aspects of programmed I/O is after outputting a character, the CPU continuously polls the device to see if it is ready to accept one.
This behavior is often called polling or Busy waiting.

```
copy_from_user(buffer,p,count); /*P is the kernel buffer*/
for(i=0;i<count;i++) {
 /* loop on every characters*/
```

```
while (*printer_status_reg!=READY); /*loop until ready*/
printer_data_register=P[i]; /*output one character */
}
return_to_user();
```

Programmed I/O is simple but has disadvantages of tying up the CPU full time until all the I/O is done. In an embedded system where the CPU has nothing else to do, busy waiting is reasonable. However in more complex system where the cpu has to do other things, busy waiting is inefficient. A better I/O method is needed.

4.5.2. Interrupt driven I/O

The problem with the programmed I/O is that the processor has to wait a long time for the input/output module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the Input/ Output module. As a result the level of performance of entire system is degraded.

An alternative approach for this is interrupt driven Input/Output. The processor issue an Input/output command to a module and then go on to do some other useful work. The input/ Output module will then interrupt the processor to request service, when it is ready to exchange data with the processor. The processor then executes the data transfer as before and then resumes its former processing. Interrupt-driven input/output still consumes a lot of time because every data has to pass with processor.

4.5.3. Direct Memory Access (DMA)

- 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.
- Handshaking between the DMA controller and the device controller is performed via a pair of wires called DMA-request and DMA-acknowledge.
- The device controller places a signal on the DMA-request wire when a word of data is available for transfer.

- This signal causes the DMA controller to seize the memory bus, place the desired address on the memory-address wires, and place a signal on the DMA-acknowledge wire.

- When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory and removes the DMA-request signal.

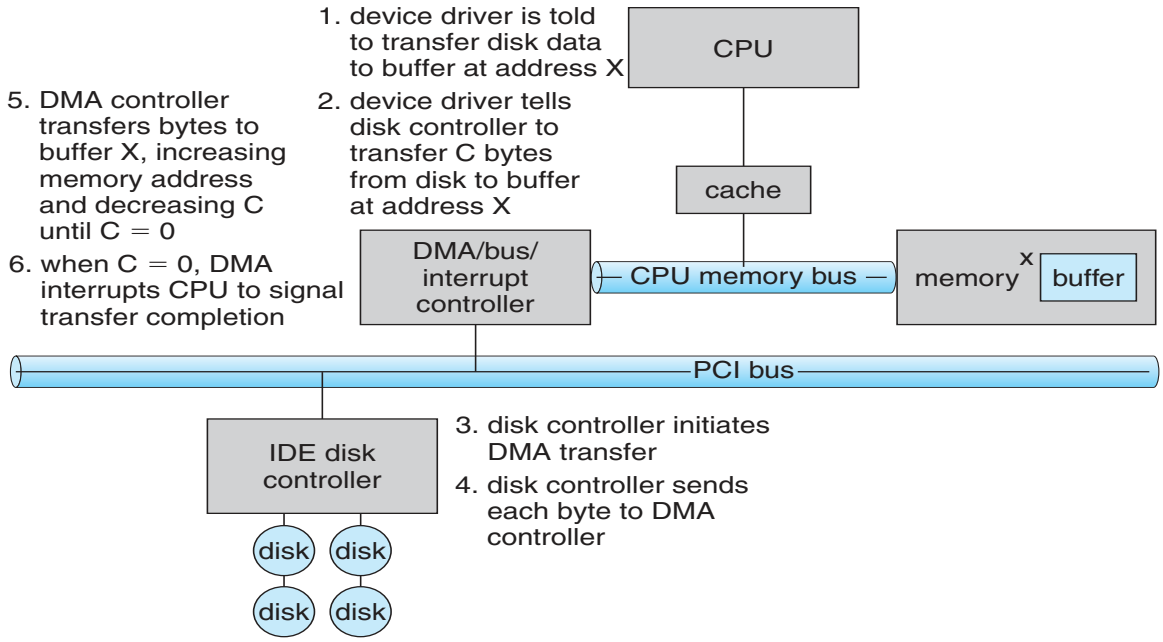


Figure 95: Six Step Process to Perform DMA Transfer

4.6. 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. Commands include read, write, and seek, Raw **I/O**, **direct I/O**, or file-system access.
    - Memory-mapped file access possible
    - File mapped to virtual memory and clusters brought via demand paging
    - DMA
    - Character devices include keyboards, mice, serial ports
    - Commands include **get()**, **put()**
    - Libraries layered on top allow line editing
- **Network Devices:** Varying enough from block and character to have own interface
    - Linux, Unix, Windows and many others include socket interface
    - Separates network protocol from network operation
    - Includes select() functionality
    - Approaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

- **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.

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Fig: Characteristic of I/O devices

4.6.1. 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.

- Provide current time, elapsed time, timer
- Normal resolution about 1/60 second
- Some systems provide higher-resolution timers
- **Programmable interval timer** used for timings, periodic interrupts
- **ioctl()** (on UNIX) covers odd aspects of I/O such as clocks and timers

4.6.2. 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.
- Implemented via multi-threading
- Returns quickly with count of bytes read or written
- **select()** to find if data ready then **read()** or **write()** to transfer

Asynchronous - process runs while I/O executes

- Difficult to use
- I/O subsystem signals process when I/O completed

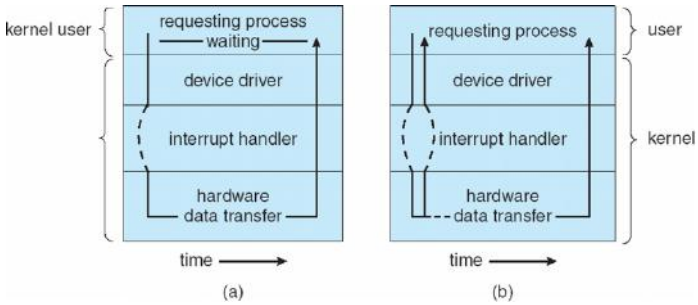


Figure 96: Two I/O model a) Synchronous b) Asynchronous

4.7. 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.

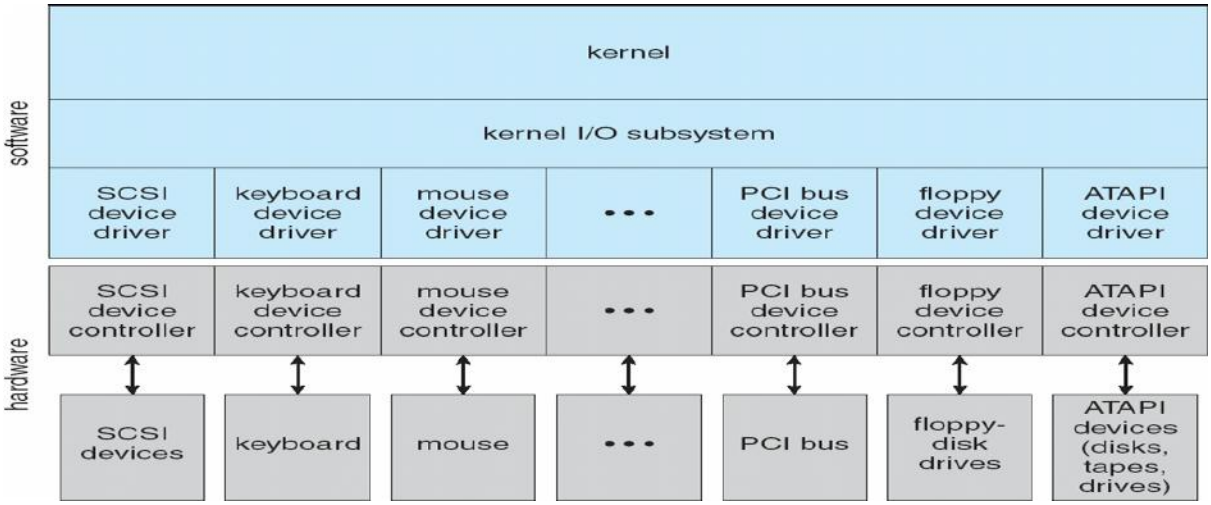


Figure 97: Kernel I/O Subsystem

4.7.1. 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.

4.7.2. Buffering

- A buffer is a memory area that stores data while they are transferred between two devices or between a devices 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.

4.7.3. 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.

4.7.4. 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.

4.7.5. Error Handling

- An operating system that uses protected memory can guard against many kinds of hardware and application errors.
- OS can recover from disk read, device unavailable, transient write failures
- Retry a read or write, for example
- Some systems more advanced – Solaris FMA, AIX
- Track error frequencies, stop using device with increasing frequency of retry-able errors
- Most return an error number or code when I/O request fails
- System error logs hold problem reports

4.7.6. 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.

4.7.7. Vectored I/O

- Vectored I/O allows one system call to perform multiple I/O operations
- For example, Unix readve() accepts a vector of multiple buffers to read into or write from
- This scatter-gather method better than multiple individual I/O calls
- Decreases context switching and system call overhead
- Some versions provide atomicity
- Avoid for example worry about multiple threads changing data as reads / writes occurring

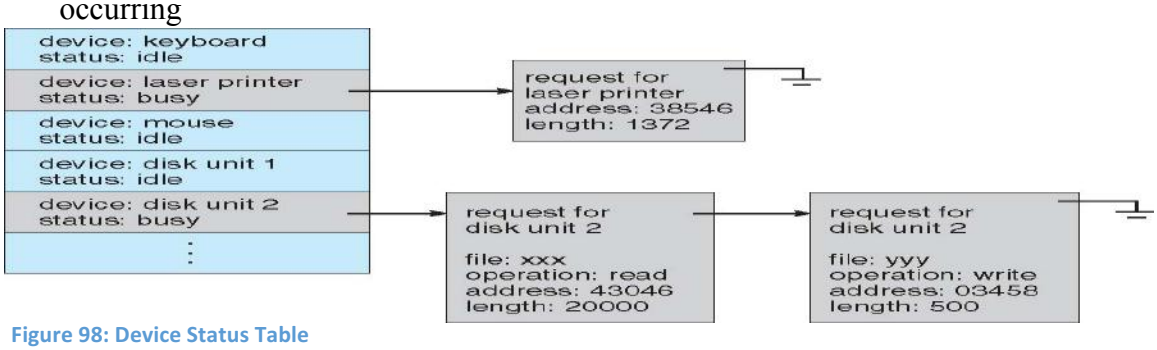


Figure 98: Device Status Table

4.7.8. I/O Protection

- User process may accidentally or purposefully attempt to disrupt normal operation via illegal I/O instructions
  - All I/O instructions defined to be privileged
    - I/O must be performed via system calls
    - Memory-mapped and I/O port memory locations must be protected too

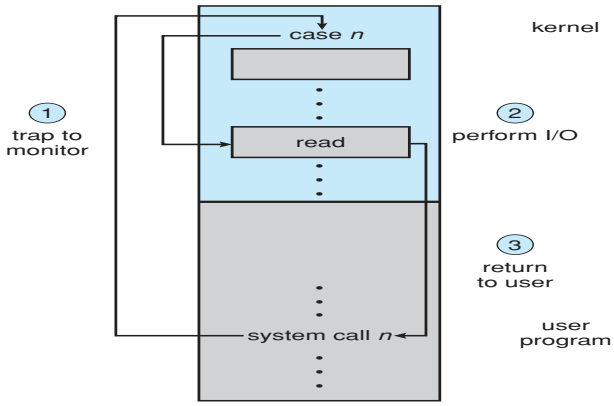


Figure 99: Use of a System Call to Perform I/O:

4.8. Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, "dirty" blocks

- Some use object-oriented methods and message passing to implement I/O
- Windows uses message passing
- Message with I/O information passed from user mode into kernel
- Message modified as it flows through to device driver and back to process
- The message-passing approach can add overhead, by comparison with procedural techniques that use shared data structures, but it simplifies the structure and design of the I/O system and adds flexibility.

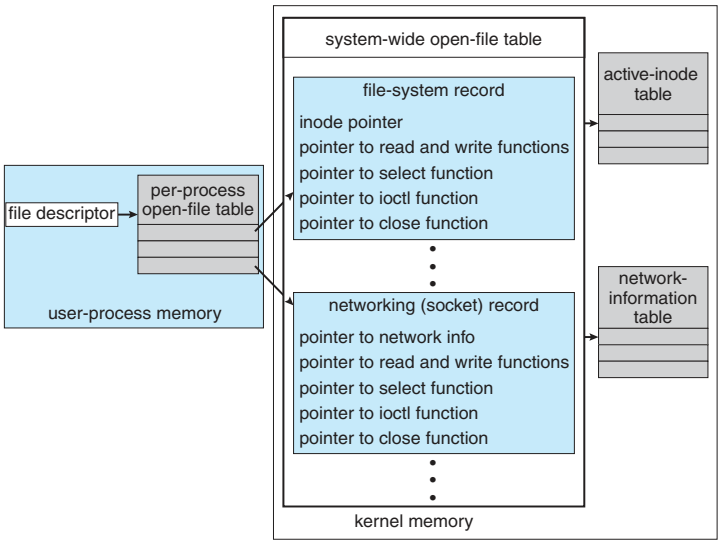


Figure 100: UNIX I/O Kernel Structure

4.9. Power Management

- Not strictly domain of I/O, but much is I/O related
- Computers and devices use electricity, generate heat, frequently require cooling
- OSes can help manage and improve use
  - Cloud computing environments move virtual machines between servers
    - Can end up evacuating whole systems and shutting them down
- Mobile computing has power management as first class OS aspect
- For example, Android implements
  - Component-level power management
    - Understands relationship between components
    - Build device tree representing physical device topology
    - System bus -> I/O subsystem -> {flash, USB storage}
    - Device driver tracks state of device, whether in use
    - Unused component – turn it off
    - All devices in tree branch unused – turn off branch
- Wake locks – like other locks but prevent sleep of device when lock is held
- Power collapse – put a device into very deep sleep
  - Marginal power use
  - Only awake enough to respond to external stimuli (button press, incoming call)

4.10 I/O Requests to Hardware Operations

- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process

- Return control to process

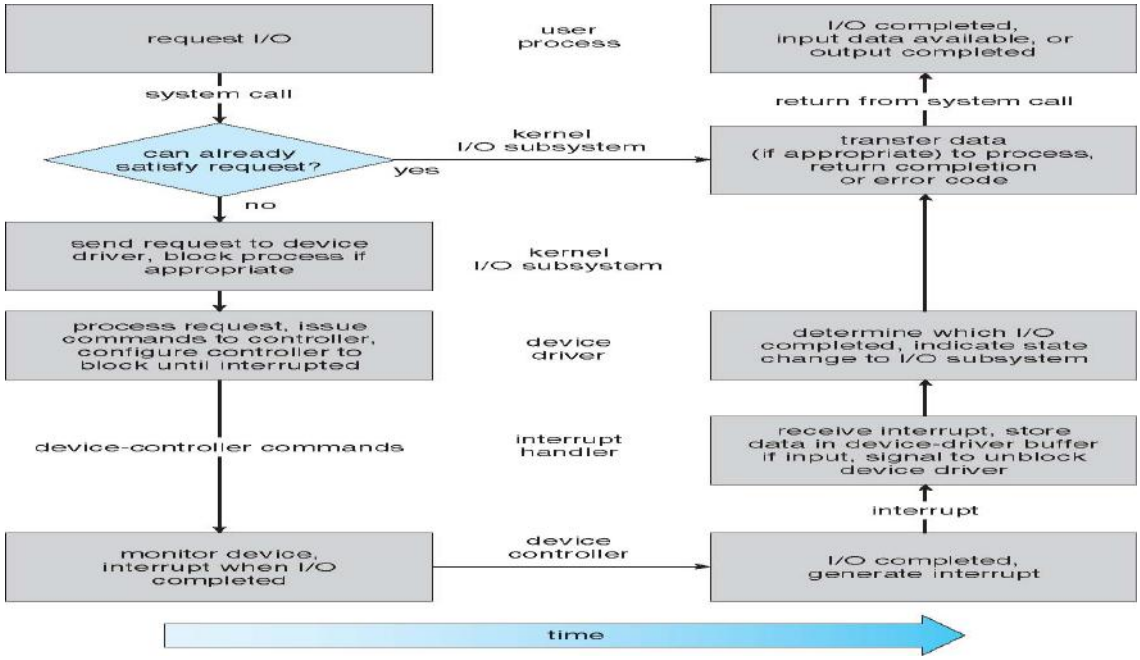


Figure 101: Life cycle of an I/O Request

4.10.1 Life Cycle of Blocking read request

1. A process issues a blocking read () system call to a file descriptor of a file that has been opened previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.
3. Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.
4. The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control register.
5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory.
7. The correct interrupt handler receives the interrupt via the interrupt vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request has completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed. ·
9. The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.

Performance

- I/O a major factor in system performance:
  - Demands CPU to execute device driver, kernel I/O code
  - Context switches due to interrupts

- Data copying
- Network traffic especially stressful

4.11 STREAMS

- STREAM – a full-duplex communication channel between a user-level process and a device in Unix System V and beyond
- A STREAM consists of:
  - STREAM head interfaces with the user process
  - driver end interfaces with the device
  - zero or more STREAM modules between them
- Each module contains a read queue and a write queue
- Message passing is used to communicate between queues
- Flow control option to indicate available or busy
- Asynchronous internally, synchronous where user process communicates with stream head

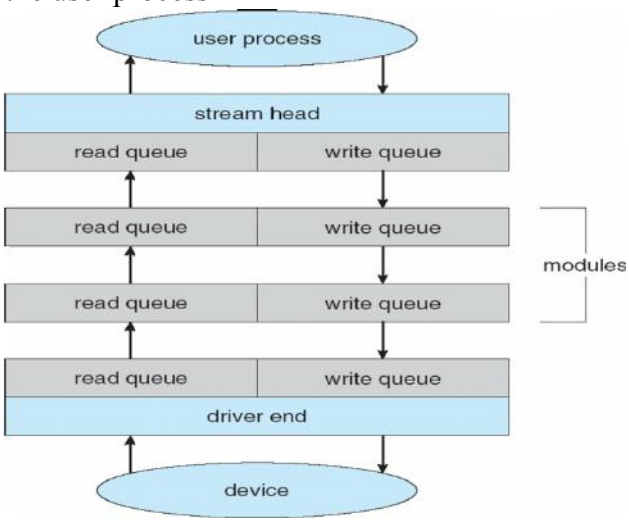


Figure 102: The STREAM Structure

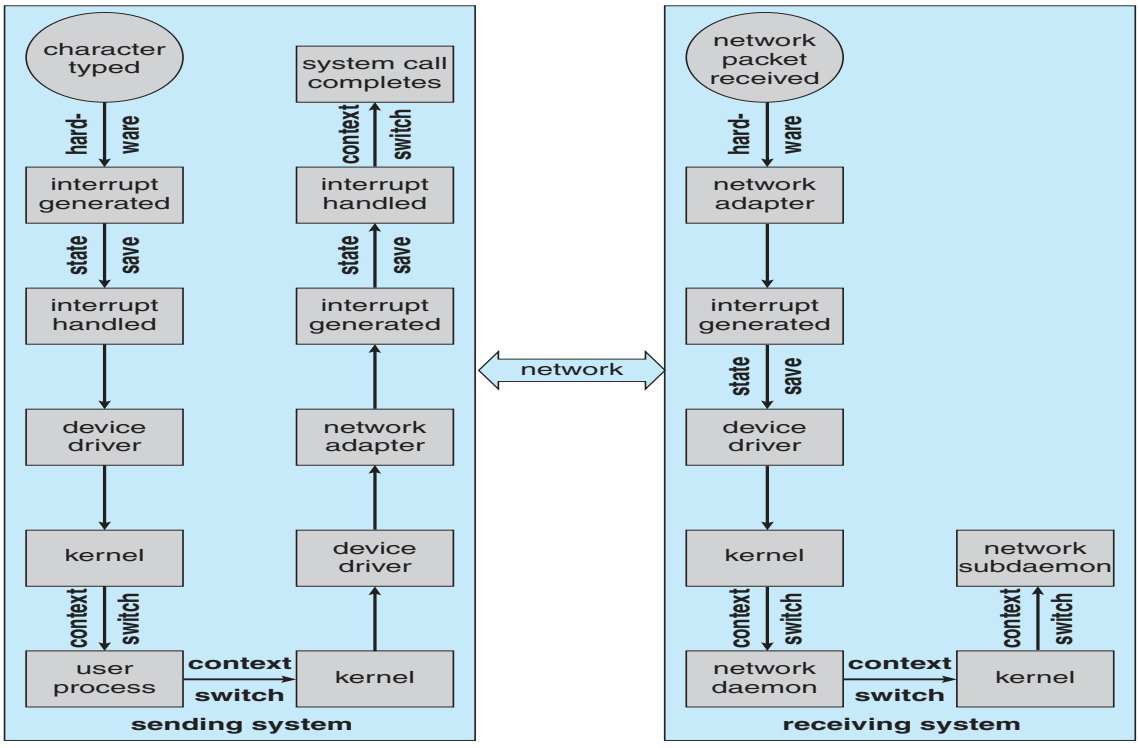


Figure 103: Intercomputer Communication

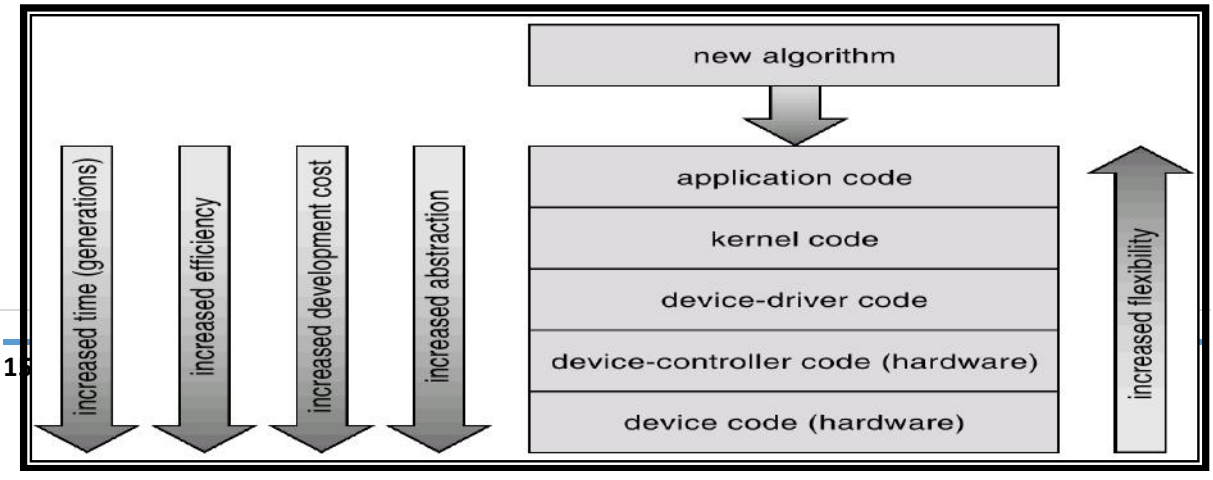
Improving Performance

Principles to improve the efficiency of I/O:

1. Reduce the number of context switches.
2. Reduce the number of times that data must be copied in memory while passing between device and application.
3. Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).
4. Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
5. Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.
6. Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

4.12. Device Functionality

- Device driver is complex.
- It not only manages individual disks but also implements RAID arrays. To do so, it converts an application's read or write request into a coordinated set of disk I/O operations.
- It implements sophisticated error-handling and data-recovery algorithms and takes many steps to optimize disk performance.
- Where the I/O functionality should be implemented- in the device hardware, in the device driver, or in application software?
- Initially, we implement experimental I/O algorithms at the application level
    - because application code is flexible and application bugs are unlikely to cause system crashes.
    - Developing code at the application level, can avoid the need to reboot or reload device drivers after every change to the code.
- But, an application-level implementation can be inefficient
    - Because of the overhead of context switches and because the application cannot take advantage of internal kernel data structures and kernel functionality (threading, locking etc.)
- Application-level algorithm may re-implement it in the kernel.
    - This can improve performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system. Moreover, an in-kernel implementation must be thoroughly debugged to avoid data corruption and system crashes.
- The highest performance may be obtained through a specialized implementation in hardware, either in the device or in the controller.
    - The disadvantages of a hardware implementation include the difficulty and expense of making further improvements or of fixing bugs, the increased development time (months rather than days), and the decreased flexibility.



4.13. Mass-Storage Device

4.13.1.Disk Structure

- Disk provide bulk of secondary storage of computer system. The disk can be considered the one I/O device that is common to each and every computer. Disks come in many size and speeds, and information may be stored optically or magnetically. Magnetic tape was used as an early secondary storage medium, but the access time is much slower than for disks. For backup, tapes are currently used.
- Modern disk drives are addressed as large one dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The actual details of disk I/O operation depends on the computer system, the operating system and the nature of the I/O channel and disk controller hardware.
- The basic unit of information storage is a sector. The sectors are stored on a flat, circular, media disk. This media spins close to one or more read/write heads. The heads can move from the inner portion of the disk to the outer portion.
- When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track. Track selection involves moving the head in a movable head system or electronically selecting one head on a fixed head system. These characteristics are common to floppy disks, hard disks, CD-ROM and DVD.
- Drives rotate at 60 to 200 times per second
- **Transfer rate** is rate at which data flow between drive and computer
- **Positioning time** (**random-access time**) is time to move disk arm to desired cylinder (**seek time**) and time for desired sector to rotate under the disk head (**rotational latency**)
- **Head crash** results from disk head making contact with the disk surface
- That's bad (no recovery, must replace disk)
- Disks can be removable
- Drive attached to computer via **I/O bus**
- Buses vary, including **EIDE, ATA, SATA, USB, Fibre Channel, SCSI**
- **Host controller** in computer uses bus to talk to **disk controller** built into drive or storage array

4.13.2.Disk Attachment

- Host-attached storage accessed through I/O ports talking to I/O buses
- SCSI itself is a bus, up to 16 devices on one cable, **SCSI initiator** requests operation and **SCSI targets** perform tasks
    - Each target can have up to 8 **logical units** (disks attached to device controller)
- Fiber Channel (FC) is high-speed serial architecture
    - Can be switched fabric with 24-bit address space – the basis of **storage area networks** (**SAN**s) in which many hosts attach to many storage units
    - Can be **arbitrated loop** (**FC-AL**) of 126 devices

4.13.3.Network-Attached Storage

- Network-attached storage (NAS) is storage made available over a network rather than over a local connection (such as a bus)

- Network File System (NFS) and Common Internet File System (CIFS) are common protocols
- Implemented via remote procedure calls (RPCs) between host and storage
- New iSCSI (latest network attached storage protocol) protocol uses IP network to carry the SCSI protocol

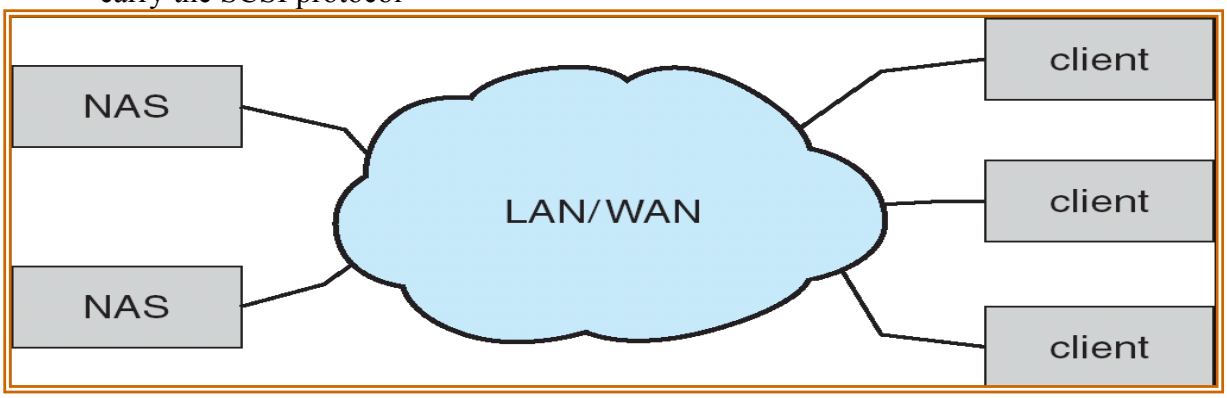


Figure 104: Networked Attached Storage

4.13.4.Storage Area Network

- One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication.
- Storage Area Network (SAN) is a private network connecting server and storage unit.
- Common in large storage environments (and becoming more common)
- Multiple hosts attached to multiple storage arrays - flexible

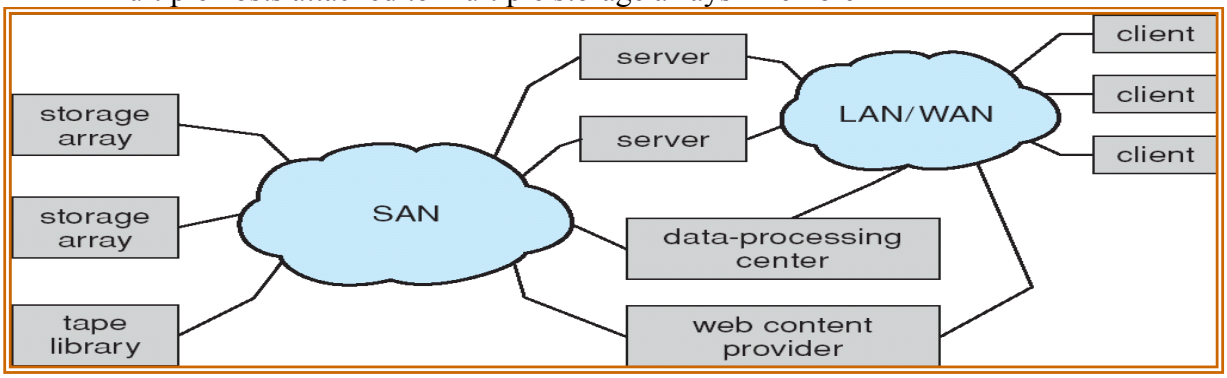


Figure 105: Storage Area Network

4.13.5.Disk Scheduling

- The operating system is responsible for using hardware efficiently for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components
  - ***Seek time*** is the time for the disk arc to move the heads to the cylinder containing the desired sector.
  - ***Rotational latency*** is the additional time waiting for the disk to rotate the desired sector to the disk head.
  - Minimize seek time
  - Seek time ≈ seek distance
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

- The amount of head needed to satisfy a series of I/O request can affect the performance. If desired disk drive and controller are available, the request can be serviced immediately. If a device or controller is busy, any new requests for service will be placed on the queue of pending requests for that drive. When one request is completed, the operating system chooses which pending request to service next.

Different types of scheduling algorithms are as follows.

1. First Come, First Served scheduling algorithm (FCFS).
2. Shortest Seek Time First (SSTF) algorithm
3. SCAN algorithm
4. Circular SCAN (C-SCAN) algorithm
5. LOOK Scheduling Algorithm
6. C-LOOK

4.13.5.1. First Come, First Served scheduling algorithm (FCFS).

- The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order. This strategy has the advantage of being fair, because every request is honored and the requests are honored in the order received. With FIFO, if there are only a few processes that require access and if many of the requests are to clustered file sectors, then we can hope for good performance.
- Priority With a system based on priority (PRI), the control of the scheduling is outside the control of disk management software.
- Last In First Out ln transaction processing systems, giving the device to the most recent user should result. In little or no arm movement for moving through a sequential file. Taking advantage of this locality improves throughput and reduces queue length.

Illustration of several algorithm with a request queue (0-199).
98, 183, 37, 122, 14, 124, 65, 67
Head pointer 53

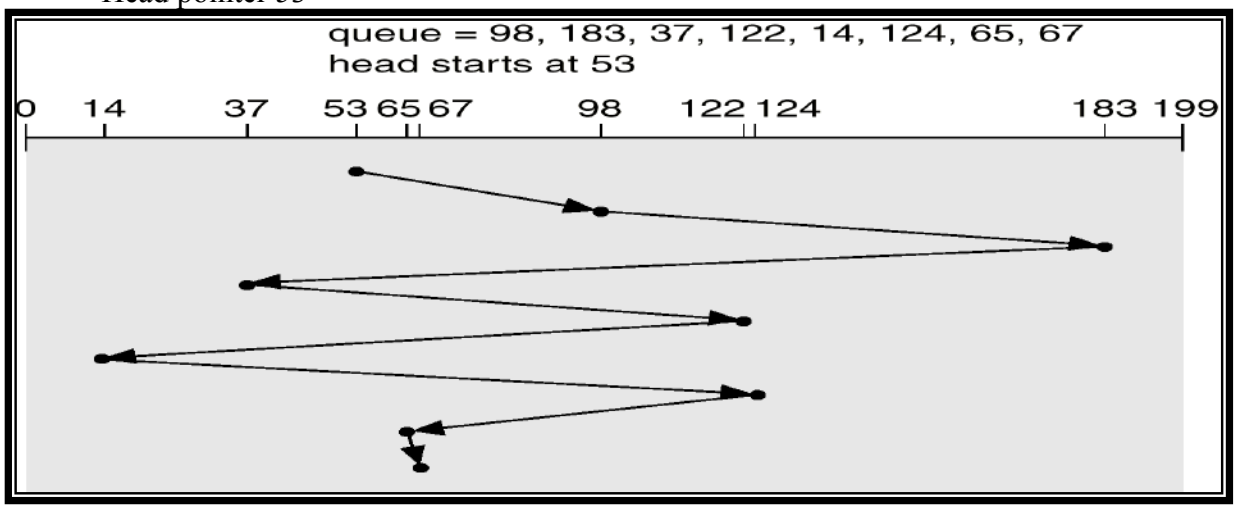


Figure 106: FIFO disk scheduling algorithm

Total head movement = 640 cylinders.

4.13.5.2. Shortest Seek Time First (SSTF) algorithm

- The SSTF policy is to select the disk I/O request the requires the least movement of the disk arm from its current position. Scan With the exception of FIFO, all of the policies described so far can leave some request unfulfilled until the entire queue is emptied. That is, there may always be new requests arriving that will be chosen before an existing request.
- The choice should provide better performance than FCFS algorithm.
- Under heavy load, SSTF can prevent distant request from ever being serviced.
- This phenomenon is known as starvation. SSTF scheduling is essentially a form of shortest job first scheduling.
- SSTF scheduling algorithm are not very popular because of two reasons.
    - Starvation possibly exists.
    - It increases higher overheads.

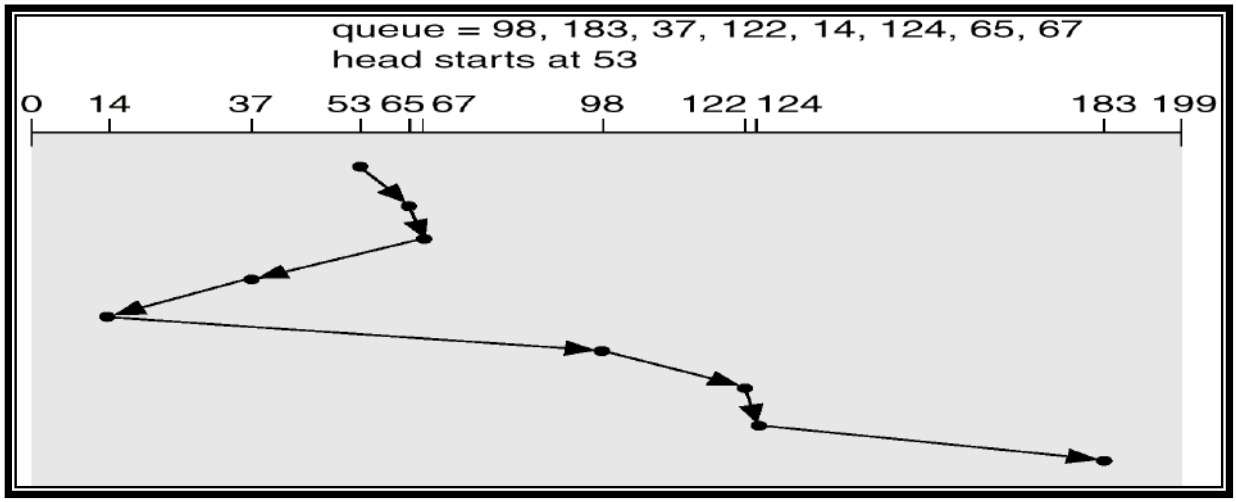


Figure 107: SSTF disk scheduling Algorithm

Total head movement = 236 cylinders.

4.13.5.3. SCAN scheduling algorithm

- The scan algorithm has the head start at track 0 and move towards the highest numbered track, servicing all requests for a track as it passes the track. The service direction is then reserved and the scan proceeds in the opposite direction, again picking up all requests in order.
- SCAN algorithm is guaranteed to service every request in one complete pass through the disk. SCAN algorithm behaves almost identically with the SSTF algorithm. The SCAN algorithm is sometimes called ***elevator*** algorithm.

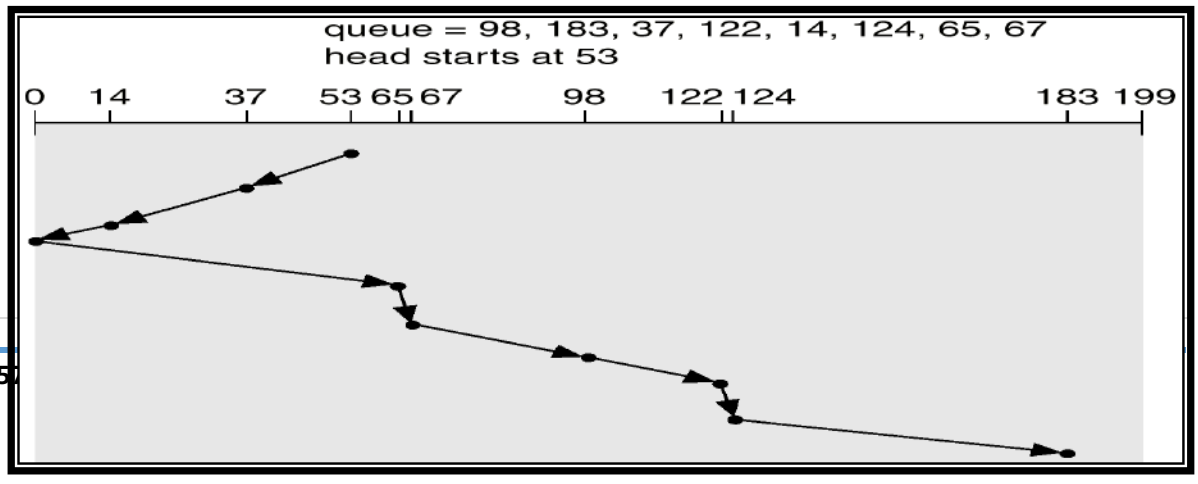


Figure 108: SCAN disk scheduling algorithm

Total head movement = 236 cylinders.

4.13.5.4. C-SCAN Scheduling Algorithm

- The C-SCAN policy restricts scanning to one direction only. Thus, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again.
- This reduces the maximum delay experienced by new requests.

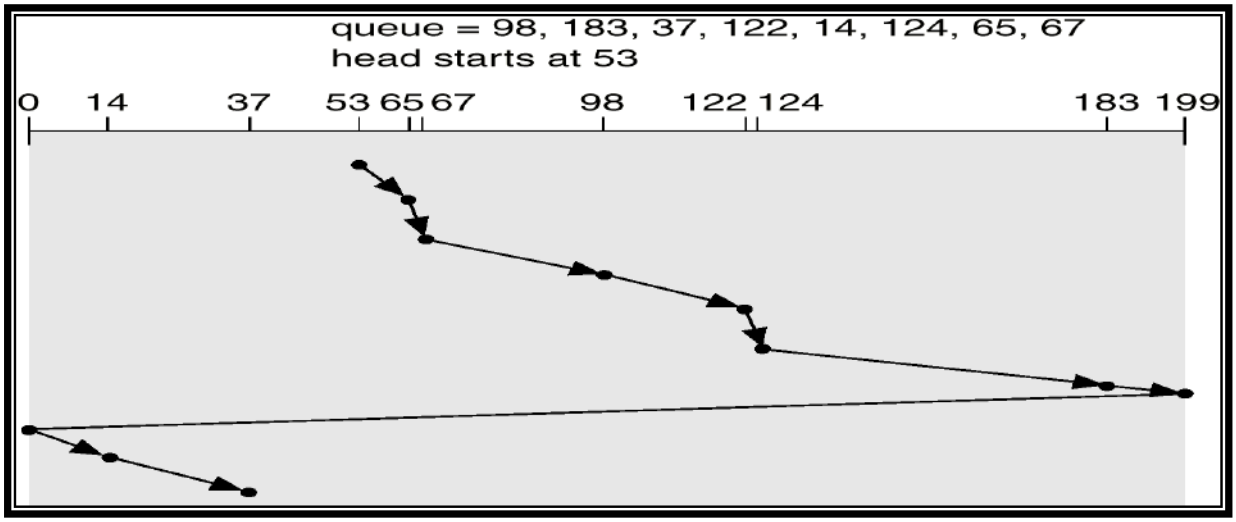


Figure 109: C-SCAN disk scheduling algorithm

4.13.5.5. LOOK Scheduling Algorithm

- Start the head moving in one direction. Satisfy the request for the closest track in that direction when there is no more request in the direction, the head is traveling, reverse direction and repeat. This algorithm is similar to innermost and outermost track on each circuit.

4.13.5.6. C-LOOK Scheduling Algorithm

- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.

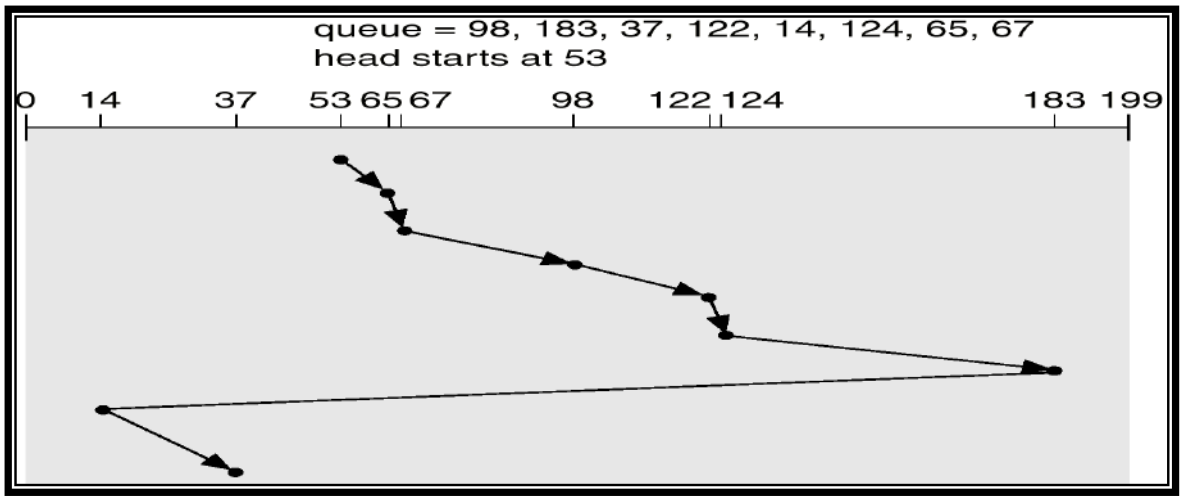


Figure 110: C-LOOK disk scheduling Algorithm

4.13.5.7. Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal.
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm.

4.13.6.Disk Management

Operating system is responsible for disk management. Following are some activities discussed.

4.13.6.1. Disk Formatting

Disk formatting is of two types.

a. Physical formatting or low level formatting.
b. Logical Formatting

Physical Formatting

- Disk must be formatted before storing data.
- Disk must be divided into sectors that the disk controllers can read/write.
- Low level formatting files the disk with a special data structure for each sector.
- Data structure consists of three fields: *header*, *data area* and *trailer*.
- Header and trailer contain information used by the disk controller.
- Sector number and Error Correcting Codes (ECC) contained in the header and trailer.
- For writing data to the sector – ECC is updated.
- For reading data from the sector – ECC is recalculated.
- If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad (**Bad Sector**).
- Low level formatting is done at factory.

Logical Formatting

- After disk is partitioned, logical formatting used.
- Operating system stores the initial file system data structures onto the disk.

4.13.6.2. Boot Block

- When a computer system is powered up or rebooted, a program in read only memory executes.
- Diagnostic check is done first.
- Stage 0 boot program is executed.
- Boot program reads the first sector from the boot device and contains a stage-1 boot program.
- May be boot sector will not contain a boot program.

- PC booting from hard disk, the boot sector also contains a partition table.
- The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code.
- Full boot strap program is more sophisticated than the bootstrap loader in the boot ROM.
- Methods such as *sector sparing* used to handle bad blocks.

4.13.6.3. Bad Blocks

- Because disks have moving parts and small tolerances, they are prone to failure.
- The disk needs to be replaced and its contents restored from backup media to the new disk.
- More frequently, one or more sectors become defective. Most disks even come from the factory with bad blocks.
- Depending on the disk and controller in use, these blocks are ***handled in a variety of ways.***

1. **Manually**
   - On simple disks, such as some disks with IDE controllers, bad blocks are handled manually.
   - Example: In MS-DOS, format command performs logical formatting and, as a part of the process, scans the disk to find bad blocks.
   - If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block.
   - If blocks go bad during normal operation, a special program (such as *chkdsk*) must be run manually to search for the bad blocks and to lock them away.
   - Data that resided on the bad blocks usually are lost.

2. **Sector Sparing or Forwarding**
   - In more sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery.
   - The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk.
   - Low-level formatting also sets aside spare sectors not visible to the operating system.
   - The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding.
   - A typical bad-sector transaction might be as follows:
     - The operating system tries to read logical block 7.
     - The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.
     - The next time the system is rebooted, a special command is run to tell the SCSI controller to replace the bad sector with a spare.
     - After that, whenever the system requests logical block 7, the request is translated into the replacement sector's address by the controller.
     - Most disks are formatted to provide a few spare sectors in each cylinder and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible.

3. **Sector Slipping**

   - Some controllers can be instructed to replace a bad block by *sector slipping*.

- Example: Suppose that logical block 7 becomes defective and the first available spare follows sector 30 and 31.
- Then, sector slipping remaps all the sectors front 7 to 30, moving them all down one spot. That is, sector 30 is copied into the spare, then sector 29 into 30, then 28 into 29, and so on, until sector 7 is copied into sector 8.
- Slipping the sectors in this way frees up the space of sector 8, so sector 7 can be mapped to it.
- The replacement of a bad block generally is not totally automatic because the data in the bad block are usually lost.

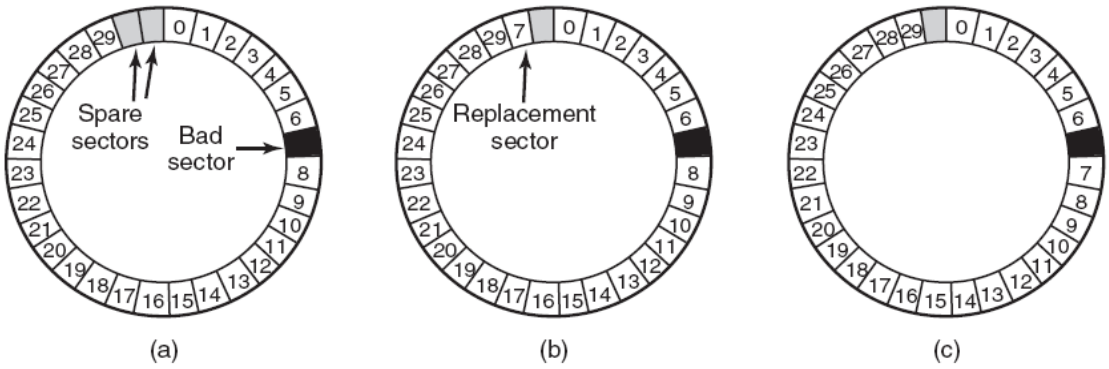


Figure 111: Bad Block Recovery (a) A disk track with a bad sector. (Sector 7 is bad) (b) Substituting a spare for the bad sector. (Sector Sparing or Forwarding (c) Shifting all the sectors to bypass the bad one. (Sector Slipping)

4.13.6.4. Interleaving

- Interleaving is a process or methodology to make a system more efficient, fast and reliable by arranging data in a non-contiguous manner. There are many uses for interleaving at the system level, including:
- Storage: As hard disks and other storage devices are used to store user and system data, there is always a need to arrange the stored data in an appropriate way.
- Interleaving is also known as sector interleave.

Why Interleaving?

- Reading continuously at very high rate requires a large buffer in the controller. **Example**: a controller with a one-sector buffer that has been given a command to read two consecutive sectors. After reading the first sector from the disk and doing the ECC calculation, the data must be transferred to main memory. While this transfer is taking place, the next sector will fly by the head. When the copy to memory is complete, the controller will have to wait almost an entire rotation time for the second sector to come around again.

Interleaved sectors

Fig: Copying to a buffer takes time; could wait a disk rotation before head reads next sector. So interleave sectors to avoid this fig (b, c)

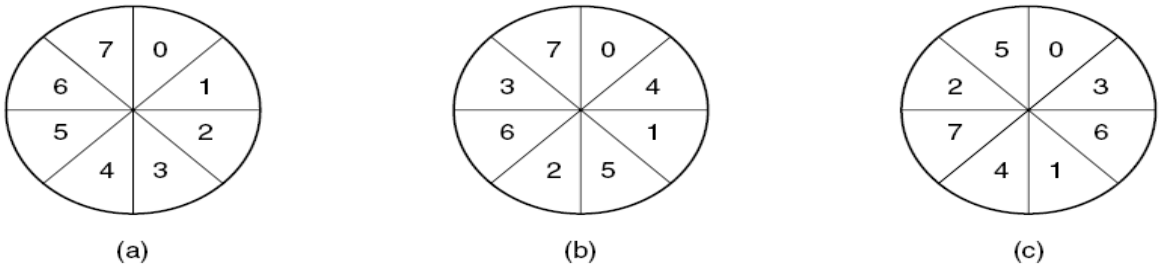


Figure 112: Interleaving (a) No Interleaving (b) Single Interleaving (c) Double Interleaving

4.13.7. Swap Space Management

Swap space management is low level task of the operating system. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system.

4.13.7.1.Swap-Space Use

The operating system needs to release sufficient main memory to bring in a process that is ready to execute. Operating system uses this swap space in various way. Paging systems may simply store pages that have been pushed out of main memory. Unix operating system allows the use of multiple swap space are usually put on separate disks, so the load placed on the I/O system by paging and swapping can be spread over the systems I/O devices.

4.13.7.2.Swap Space Location

Swap space can reside in two places:

1. Separate disk partition
2. Normal file System

- If the swap space is simply a large file within the file system, normal file system routines can be used to create it, name it and allocate its space. This is easy to implement but also inefficient. External fragmentation can greatly increase swapping times. Catching is used to improve the system performance. Block of information is cached in the physical memory, and by using special tools to allocate physically continuous blocks for the swap file.
- Swap space can be created in a separate disk partition. No file system or directory structure is placed on this space. A separate swap space storage manager is used to allocate and deallocate the blocks. This manager uses algorithms optimized for speed. Internal fragmentation may increase. Some operating systems are flexible and can swap both in raw partitions and in file system space.

4.13.8. Stable Storage Implementation

- The write ahead log, which required the availability of stable storage.
- By definition, information residing in stable storage is never lost.
- To implement such storage, we need to replicate the required information on multiple storage devices (usually disks) with independent failure modes.
- We also need to coordinate the writing of updates in a way that guarantees that a failure during an update will not leave all the copies in a damaged state and that, when we are recovering from failure, we can force all copies to a consistent and correct value, even if another failure occurs during the recovery.

4.13.8.1.Tertiary Storage Devices

- Low cost is the defining characteristic of tertiary storage.
- Generally, tertiary storage is built using *removable media*
- Common examples of removable media are floppy disks and CD-ROMs; other types are available.

4.13.8.2.Removable Disks

- Floppy disk — thin flexible disk coated with magnetic material, enclosed in a protective plastic case.
    - Most floppies hold about 1 MB; similar technology is used for removable disks that hold more than 1 GB.
    - Removable magnetic disks can be nearly as fast as hard disks, but they are at a greater risk of damage from exposure.
- A magneto-optic disk records data on a rigid platter coated with magnetic material.
    - Laser heat is used to amplify a large, weak magnetic field to record a bit.
    - Laser light is also used to read data (Kerr effect).
    - The magneto-optic head flies much farther from the disk surface than a magnetic disk head, and the magnetic material is covered with a protective layer of plastic or glass; resistant to head crashes.
- Optical disks do not use magnetism; they employ special materials that are altered by laser light.

4.13.8.3.WORM Disks

- The data on read-write disks can be modified over and over.
- WORM ("Write Once, Read Many Times") disks can be written only once.
- Thin aluminum film sandwiched between two glass or plastic platters.
- To write a bit, the drive uses a laser light to burn a small hole through the aluminum; information can be destroyed by not altered.
- Very durable and reliable.
- *Read Only* disks, such as CD-ROM and DVD, come from the factory with the data pre-recorded.

4.13.8.4.Tapes

- Compared to a disk, a tape is less expensive and holds more data, but random access is much slower.
- Tape is an economical medium for purposes that do not require fast random access, e.g., backup copies of disk data, holding huge volumes of data.
- Large tape installations typically use robotic tape changers that move tapes between tape drives and storage slots in a tape library.
    - stacker – library that holds a few tapes
    - silo – library that holds thousands of tapes
- A disk-resident file can be *archived* to tape for low cost storage; the computer can *stage* it back into disk storage for active use.

4.13.8.5.Disk Reliability

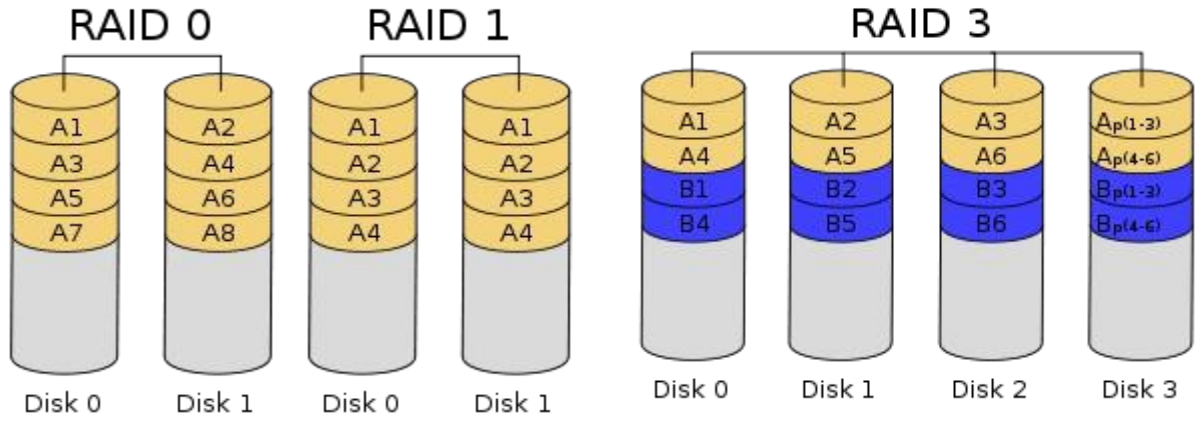
- Good performance means high speed, another important aspect of performance is reliability.
- A fixed disk drive is likely to be more reliable than a removable disk or tape drive.
- An optical cartridge is likely to be more reliable than a magnetic disk or tape.
- A head crash in a fixed hard disk generally destroys the data, whereas the failure of a tape drive or optical disk drive often leaves the data cartridge unharmed.

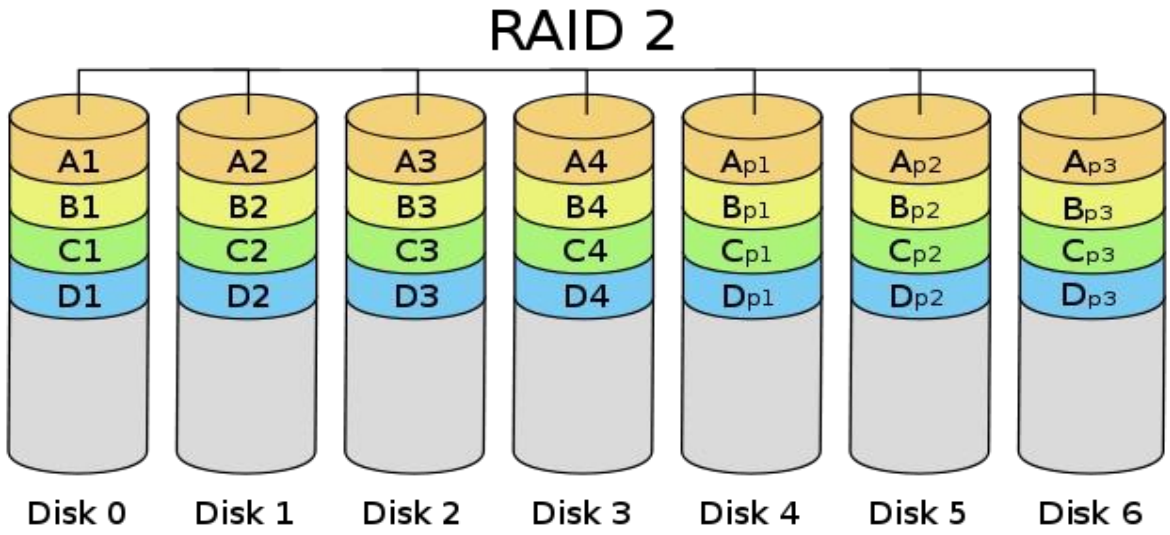
4.13.8.6.RAID (Redundant Array of Independent Disk)

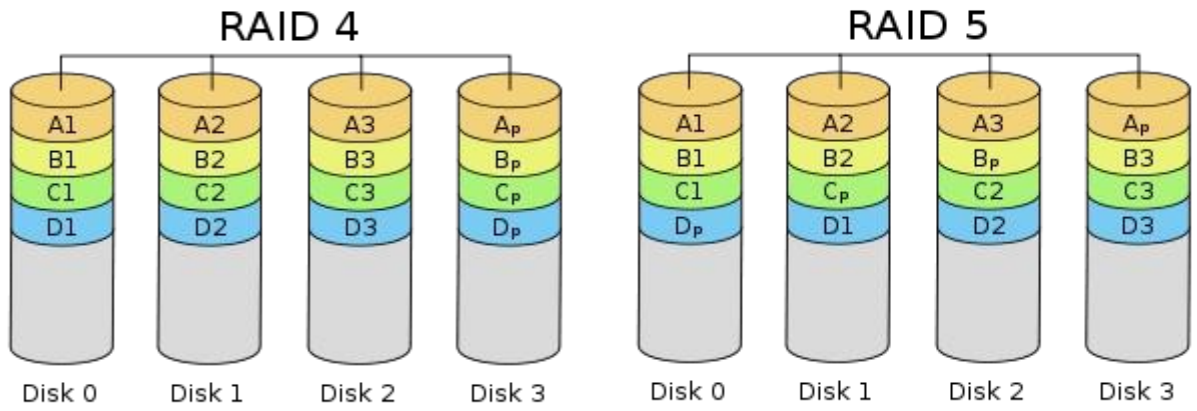
RAID (redundant array of independent disks), originally redundant array of inexpensive disks is a storage technology that combines multiple disk drive components into a logical unit. Data is distributed across the drives in one of several ways called "RAID levels", depending on what level of redundancy and performance (via parallel communication) is required.

RAID Levels

RAID 0 (block-level striping without parity or mirroring) has no (or zero) redundancy. It provides improved performance and additional storage but no fault tolerance. Hence simple stripe sets are normally referred to as RAID 0. Any drive failure destroys the array, and the likelihood of failure increases with more drives in the array (at a minimum, catastrophic data loss is almost twice as likely compared to single drives without RAID). A single drive failure destroys the entire array because when data is written to a RAID 0 volume, the data is broken into fragments called blocks. The number of blocks is dictated by the stripe size, which is a configuration parameter of the array. The blocks are written to their respective drives simultaneously on the same sector. This allows smaller sections of the entire chunk of data to be read off each drive in parallel, increasing bandwidth. RAID 0 does not implement error checking, so any error is uncorrectable. More drives in the array means higher bandwidth, but greater risk of data loss







In **RAID 1** (mirroring without parity or striping), data is written identically to multiple drives, thereby producing a "mirrored set"; at least 2 drives are required to constitute such an array. While more constituent drives may be employed, many implementations deal with a maximum of only 2; of course, it might be possible to use such a limited level 1 RAID itself as a constituent of a level 1 RAID, effectively masking the limitation. The array continues to operate as long as at least one drive is functioning. With appropriate operating system support, there can be increased read performance, and only a minimal write performance reduction; implementing RAID 1 with a separate controller for each drive in order to perform simultaneous reads (and writes) is sometimes called multiplexing (or duplexing when there are only 2 drives).

In **RAID 2** (bit-level striping with dedicated Hamming-code parity), all disk spindle rotation is synchronized, and data is striped such that each sequential bit is on a different drive. Hamming-code parity is calculated across corresponding bits and stored on at least one parity drive.

In **RAID 3** (byte-level striping with dedicated parity), all disk spindle rotation is synchronized, and data is striped so each sequential byte is on a different drive. Parity is calculated across corresponding bytes and stored on a dedicated parity drive.

RAID 4 (block-level striping with dedicated parity) is identical to RAID 5 (see below), but confines all parity data to a single drive. In this setup, files may be distributed between multiple drives. Each drive operates independently, allowing I/O requests to be performed in parallel. However, the use of a dedicated parity drive could create a performance bottleneck; because the parity data must be written to a single, dedicated parity drive for each block of non-parity data, the overall write performance may depend a great deal on the performance of this parity drive.

RAJD 5 (block-level striping with distributed parity) distributes parity along with the data and requires all drives but one to be present to operate; the array is not destroyed by a single drive failure. Upon drive failure, any subsequent reads can be calculated from the distributed parity such that the drive failure is masked from the end user. However, a single drive failure results in reduced performance of the entire array until the failed drive has been replaced and the associated data rebuilt. Additionally, there is the potentially disastrous RAID 5 write hole. RAID 5 requires at least 3 disks.

RAID 6 (block-level striping with double distributed parity) provides fault tolerance of two drive failures; the array continues to operate with up to two failed drives. This makes larger

RAID groups more practical, especially for high-availability systems. This becomes increasingly important as large- capacity drives lengthen the time needed to recover from the failure of a single drive. Single-parity RAID levels are as vulnerable to data loss as a RAID 0 array until the failed drive is replaced and its data rebuilt; the larger the drive, the longer the rebuild takes. Double parity gives additional time to rebuild the array without the data being at risk if a single additional drive fails before the rebuild is complete.

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.
- 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.
- 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.
- Disk drives are the major secondary storage I/O devices on most computers. Most secondary devices are either magnetic disks or magnetic tapes. Modern disk drives are structured as large one dimensional arrays of logical disk blocks. Disk scheduling algorithms can improve the effective bandwidth, the average response time, and the variance response time. Algorithms such as SSTF, SCAN, C-SCAN, LOOK, and CLOOK are designed to make such improvements through strategies for disk queue ordering.
- Performance can be harmed by external fragmentation. The operating system manages block. First, a disk must be low level formatted to create the sectors on the raw hardware, new disks usually come preformatted. Then, disk is partitioned, file systems are created, and boot blocks are allocated to store the system bootstrap program. Finally when a block is corrupted, the system must have a way to lock out that block or to replace it logically with a space.
- Because of efficient swap space is a key to good performance, systems usually bypass the file system and use raw disk access for paging I/O. Some systems dedicate a raw disk partition to swap space, and others use a file within the file system instead.
- The write-ahead log scheme requires the availability of stable storage. Tertiary storage is built from disk and tape drives that use removable media. Many different technologies are available, including magnetic tape, removable magnetic and magneto-optic disks, and optical disks. Three important aspects of performance are bandwidth, latency, and reliability. Many bandwidths are available for both disks and tapes, but the random-access latency for a tape is generally much greater than that for a disk.

CHAPTER 5

FILE SYSTEMS

5.1.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 user's 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.
  1.Text File
  2.Source File
  3.Executable File
  4.Object File

5.1.1. File Structure

Three terms are used for files

- 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 single 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.

5.1.2. 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

5.1.3. 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.

5.1.4. 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 or none	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 languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Table 9: File Types

5.1.5. 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 are the 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.

5.1.6. 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.

5.1.7. 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.

5.1.8. Access Methods

- Sequential Access
    - *read next*
    - *write next*
    - *reset*
    - no *read after last write*
        - *(rewrite)*
- Direct Access
    - *read n*
    - *write n*
    - *position to n*
        - *read next*
        - *write next*
    - *rewrite n*
    - *n* = relative block number

5.1.8.1. Sequential-access File

- Information in the file is processed in order, one record after the other.
- Reads and writes make up the bulk of the operations on a file.
- A read operation-read *next-reads* the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- The write *operation-write* next-appends to the end of the file and advances to the end of the newly written material (the new end of file)

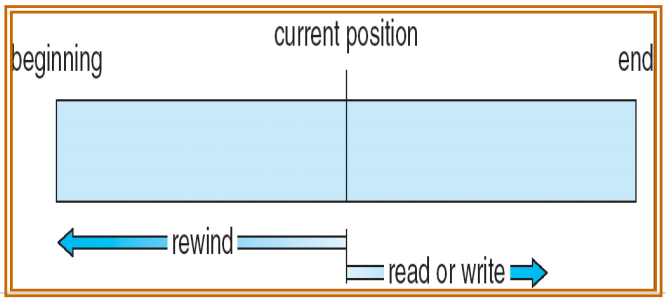


Figure 113: Sequential access of file

sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Figure 114: Simulation of Sequential Access on a Direct-access File

5.1.8.2. Other Access Methods

- Can be built on top of base methods
- General involve creation of an index for the file
- Keep index in memory for fast determination of location of data to be operated on
- If too large, index (in memory) of the index (on disk)
- IBM indexed sequential-access method (ISAM)
    - Small master index, points to disk blocks of secondary index
    - File kept sorted on a defined key
    - All done by the OS

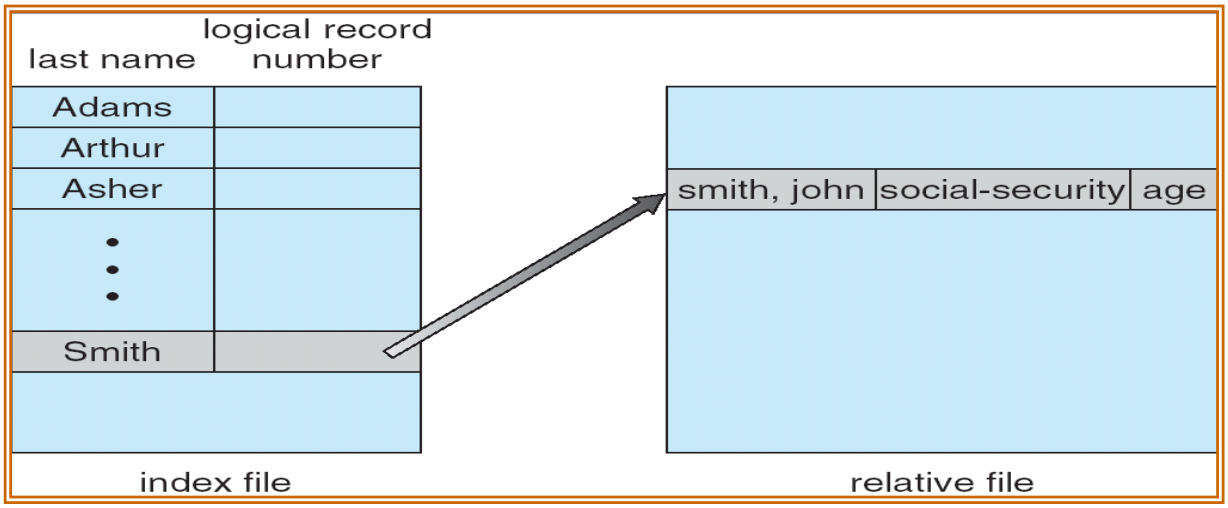


Figure 115: Example of Index and Relative Files

5.2.Directory Structure

- The file system of computers can be quite extensive.
- Because Some systems store thousands of files on hundreds of gigabytes of disk
- To manage all these files we need to organize them.
- This organization is usually done in two parts
    - First the file system is broken into partitions (known as minidisks in the IBM world and volumes in the pc and Macintosh)
        - Each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside.

- Second each partitions contains information about files within it. This information is kept in entries in a device directory or volume table of contents.
  - The device directory records information such as name, location, size and type for all the files on that partition

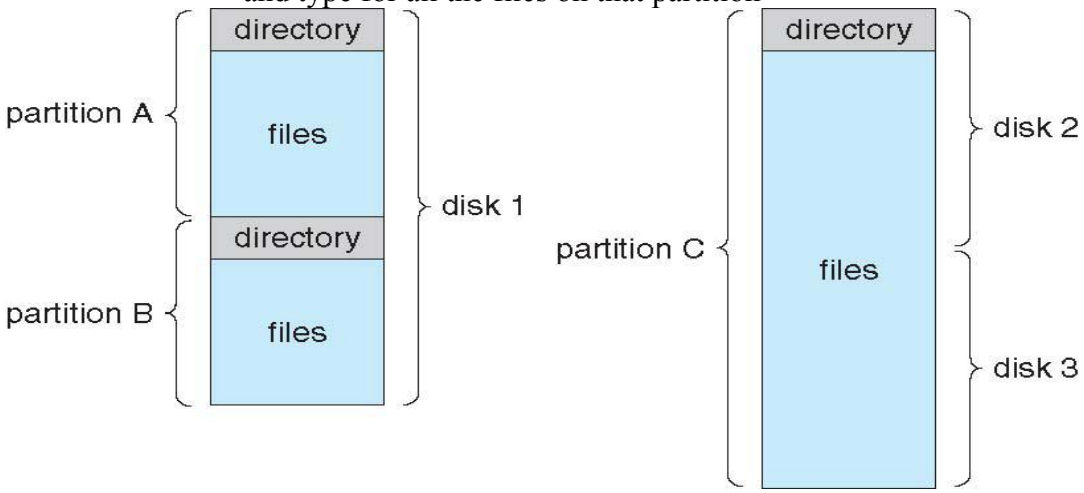


Figure 116: A typical File system organization

5.2.1. Information in a Device Directory

- Name
- Type
- Address
- Current length
- Maximum length
- Date last accessed (for archival)
- Date last updated (for dump)
- Owner ID (who pays)
- Protection information

5.2.2. Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system

5.2.3. Organize the Directory (Logically) to Obtain

- Efficiency – locating a file quickly.
- Naming – convenient to users.
  - Two users can have same name for different files.
  - The same file can have several different names.
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, …)
- The most common scheme for defining the logical structure of a directory are:
  - single-level directory

- two-level directory
- tree-structured directories
- Acyclic-graph directories
- General graph directory

5.2.3.1. Single-Level Directory

- A single directory for all users.
- Naming problem
- Grouping problem(user wise)

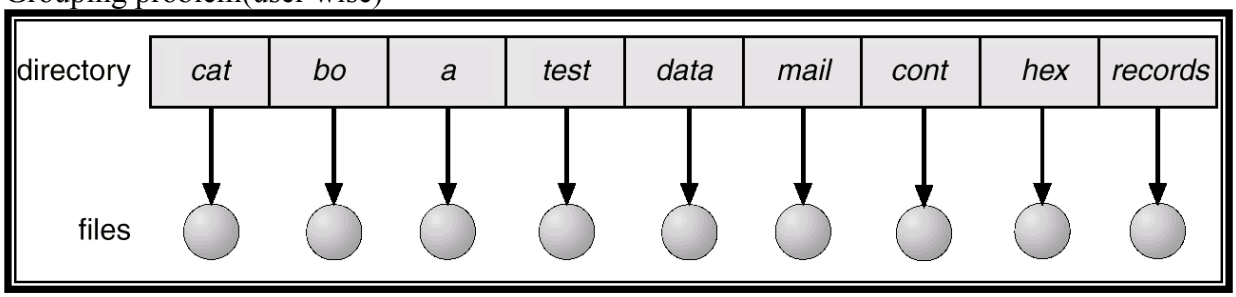


Figure 117: Single Level Directory

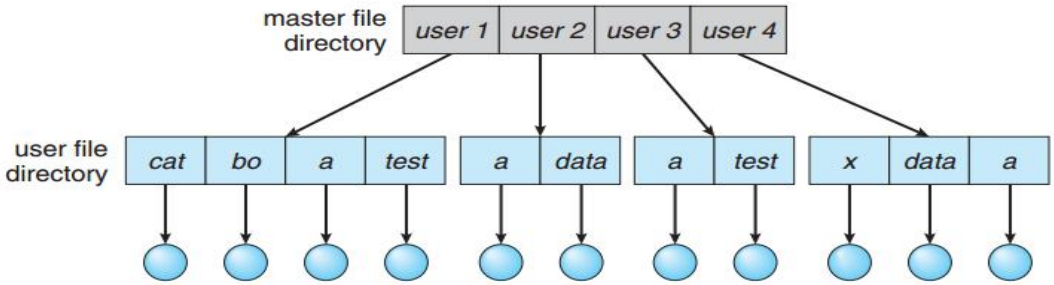
5.2.3.2. Two-level directory

- The limitation of single-level directory structure is the confusion of filenames created by different users.
- The standard solution is to create a separate directory for each user.
- This solution is used by two level directory structure.
- In two-level directory structure each user has his/her own directory called user file directory(UFD)
- Each UFD has a similar structure, but lists only the files of a single user.
- In a system with two level directory structure, when a user job starts or user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by username or account number and each entry points to the UFD for that user.
    - ***Search a file***: when a user refers to a particular file only his/her own UFD is searched. Thus different user may have files with the same name, as long as all the file names within each UFD are unique.
    - ***Create a file***: to create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists.
    - ***Delete a file***: to delete a file the operating system confines its search to the local UFD, thus, it cannot accidently delete another user's file that has the same name.
- In two level directory the user directories themselves must be created and deleted as necessary(i.e when new user is created and when existing user is deleted)
- For above purpose a special program is run with the appropriate username and account information.
- This program creates a new user file directory(UFD) and adds an entry for it to the master file directory (MFD)
- The execution of this program might be restricted to system administrators. The allocation of disk space for user directories can be handled by different techniques.

Problems:

- The two-level directory structure isolates one user from another.

- This isolation is an advantage when the users are completely independent but is disadvantage when the users want to cooperate on some task and to access one



another's files.

Figure 118: Two Level Directoory

- A two level directory structure can be thought of as a tree or an inverted tree of height two as shown in figure below:
- The root of the tree is the MFD.
- MFD's direct descendants are UFD's
- The descendants of UFD's are the files themselves
- The files are leaves of the tree.
- Specifying a username and file name defines a path in the tree from root(directory) to the leaf(file)
- Thus username and filename define a **pathname**
- **Every file in the system has a path name.**
- There is additional syntax to specify the partition of a file.
- For example, in MS-DOS a partition is specified by a letter followed by colon. Thus, a file specification might be c:\userb\test.

5.2.3.3. Tree structured directories

- The two level directory structure does not allow user to create their own directories
- The tree structure allows users to create their own subdirectories and to organize their files accordingly.
- E.g. MS-DOS
- Tree is the most common directory structure
- A directory contains a set of files or subdirectories
- A directory is simply another file, but it is treated in a special way.
- One bit in directory entry defines the entry as a file (0) or as a subdirectory (1)
- Special system calls create and delete directories.
- In normal use, each user has a **current directory**. The current directory contains most of the files that are of current interest to the user.
- Initial current directory of a user is designated when the user job starts or the user logs in.
- Path names can be of two types:
  - Absolute path: absolute path begins at the root and follows a path down to the specified file

- Relative path: relative path defines a path from the current directory

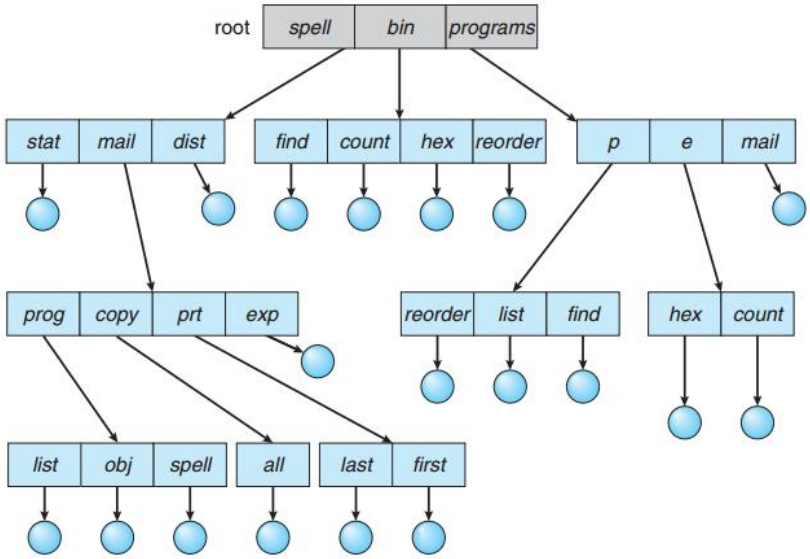


Figure 119: Tree structured directories

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - **cd** /spell/mail/prog
  - **type** list

5.2.4. Acyclic graph directories

- Consider a situation when two programmer are working on a joint project.
- In such situation a common subdirectory should be shared
- A shared directory or file will exist in the file system in two or more places at once.
- Note that a shared file or directory is not the same as two copies of the file.
- In case of shared file, there is only one actual file
- A tree structure prohibits the sharing of files and directories
- Acyclic graph directory structure allows directories to have shared subdirectories and files as shown in figure below:

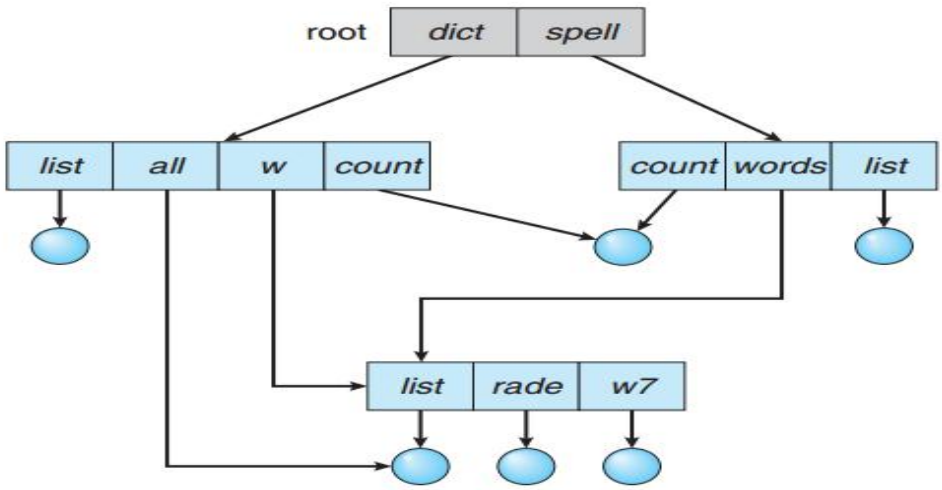


Figure 120: Acyclic graph directories

5.2.5 File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a protection scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- If multi-user system
    - **User IDs** identify users, allowing permissions and protections to be per-user
    - **Group IDs** allow users to be in groups, permitting group access rights
- Owner of a file / directory
- Group of a file / directory

5.2.6. Remote File Systems

- Uses networking to allow file system access between systems
    - Manually via programs like FTP
    - Automatically, seamlessly using distributed file systems
    - Semi automatically via the world wide web
- Client-server model allows clients to mount remote file systems from servers
    - Server can serve multiple clients
    - Client and user-on-client identification is insecure or complicated
    - NFS is standard UNIX client-server file sharing protocol
    - CIFS is standard Windows protocol
    - Standard operating system file calls are translated into remote calls
- Distributed Information Systems (distributed naming services) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing.

5.2.7. Failure Modes

- All file systems have failure modes
    - For example corruption of directory structures or other non-user data, called metadata
- Remote file systems add new failure modes, due to network failure, server failure
- Recovery from failure can involve state information about status of each remote request
- Stateless protocols such as NFS v3 include all information in each request, allowing easy recovery but less security

5.2.8. Consistency Semantics

- Specify how multiple users are to access a shared file simultaneously
    - Similar process synchronization algorithms
        - Tend to be less complex due to disk I/O and network latency (for remote file systems)
    - Andrew File System (AFS) implemented complex remote file sharing semantics

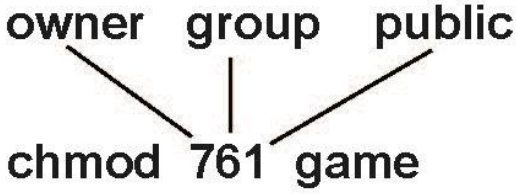
- Unix file system (UFS) implements:
  - Writes to an open file visible immediately to other users of the same open file
  - Sharing file pointer to allow multiple users to read and write concurrently
- AFS has session semantics
  - Writes only visible to sessions starting after the file is closed

5.2.9. Protection

- File owner/creator should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List
- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

			RWX
a) **owner access**	7	⇒	1 1 1
b) **group access**	6	⇒	RWX 1 1 0
c) **public access**	1	⇒	RWX 0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.
- Attach a group to a file *chgr*



5.3. File System Implementation

- We have system calls at the API level, but how do we implement their functions?
  - On-disk and in-memory structures
- Boot control block contains info needed by system to boot OS from that volume
  - Needed if volume contains OS, usually first block of volume
- Volume control block (superblock, master file table) contains volume details
  - Total # of blocks, # of free blocks, block size, free block pointers or array
- Directory structure organizes the files
  - Names and inode numbers, master file table
- Per-file **File Control Block (FCB)** contains many details about the file
  - inode number, permissions, size, dates

- NTFS stores into in master file table using relational DB structures

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

In-Memory File System Structures

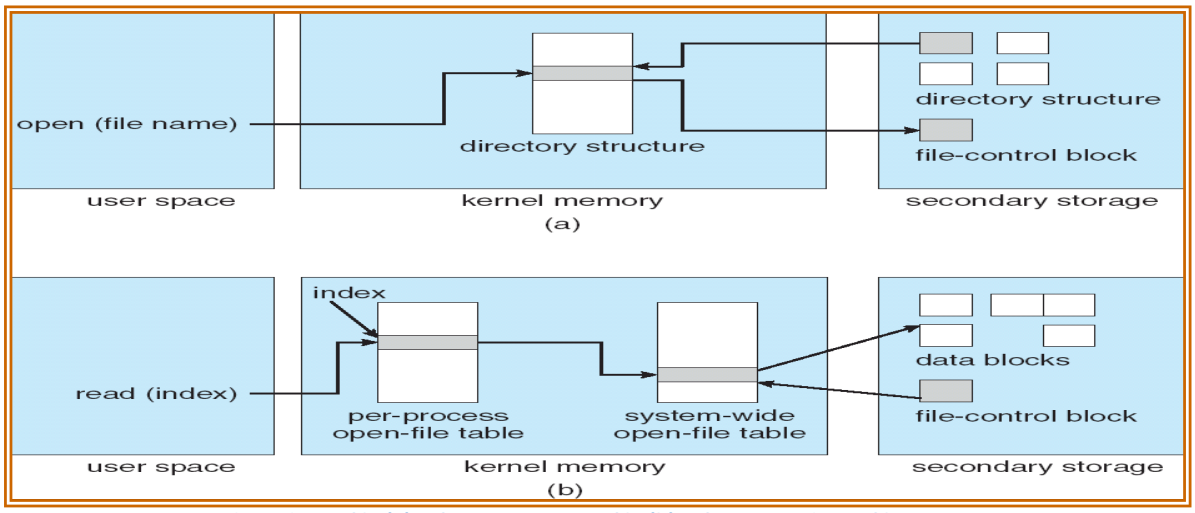


Figure 121: Opening and Reading a file (a) refers to opening a file (b) refers to reading a file

5.3.1. Partitions and Mounting

- Partition can be a volume containing a file system ("cooked") or raw – just a sequence of blocks with no file system
- Boot block can point to boot volume or boot loader set of blocks that contain enough code to know how to load the kernel from the file system
  - Or a boot management program for multi-os booting
- Root partition contains the OS, other partitions can hold other OSes, other file systems, or be raw
  - Mounted at boot time
  - Other partitions can mount automatically or manually
- At mount time, file system consistency checked
  - Is all metadata correct?
    - If not, fix it, try again
    - If yes, add to mount table, allow access

5.3.2. Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.

- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

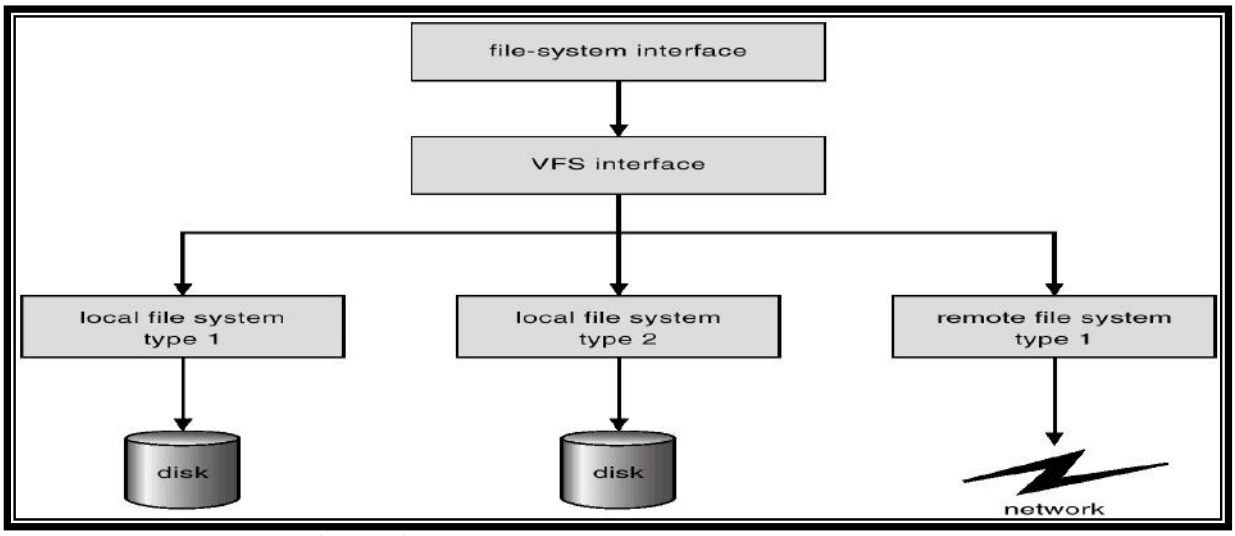


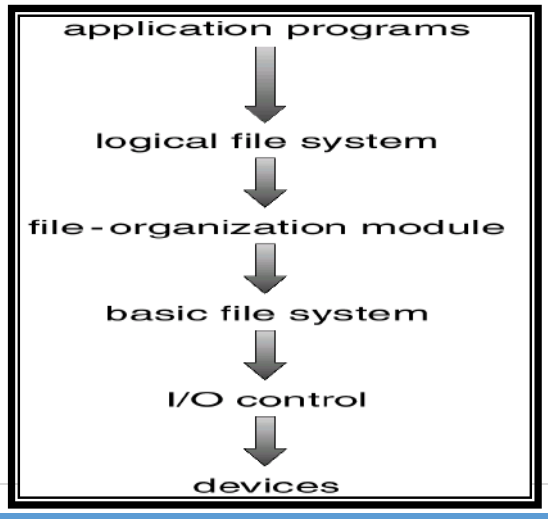
Figure 122: Systematic view of virtual file system

5.3.3. File-System Structure

- File structure
  - Logical storage unit
  - Collection of related information
- **File system** resides on secondary storage (disks)
  - Provided user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily
- Disk provides in-place rewrite and random access
  - I/O transfers performed in **blocks** of **sectors** (usually 512 bytes)
- **File control block** – storage structure consisting of information about a file
- **Device driver** controls the physical device
- File system organized into layers
- **Boot control block** contains info needed by system to boot OS from that volume
- **Volume control block** (*super block*) contains volume details
- Directory structure organizes the files
- Per-file File Control Block (FCB) contains many details about the file (*inode*)

5.3.4. Layer File System

- Device drivers manage I/O devices at the I/O control layer
  - Given commands like "read drive1, cylinder 72, track 2, sector 10, into memory location 1060" outputs low-level hardware specific commands to hardware controller
- Basic file system given command like "retrieve block 123" translates to device driver



- Also manages memory buffers and caches (allocation, freeing, replacement) Fig: Layer File System
  - Buffers hold data in transit
  - Caches hold frequently used data
- File organization module understands files, logical address, and physical blocks
  - Translates logical block # to physical block #
  - Manages free space, disk allocation
- **Logical file system** manages metadata information
  - Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Directory management
  - Protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance. Translates file name into file number, file handle, location by maintaining file control blocks (**inodes** in UNIX)
  - Logical layers can be implemented by any coding method according to OS designer

5.4. File 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.

5.4.1. 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.

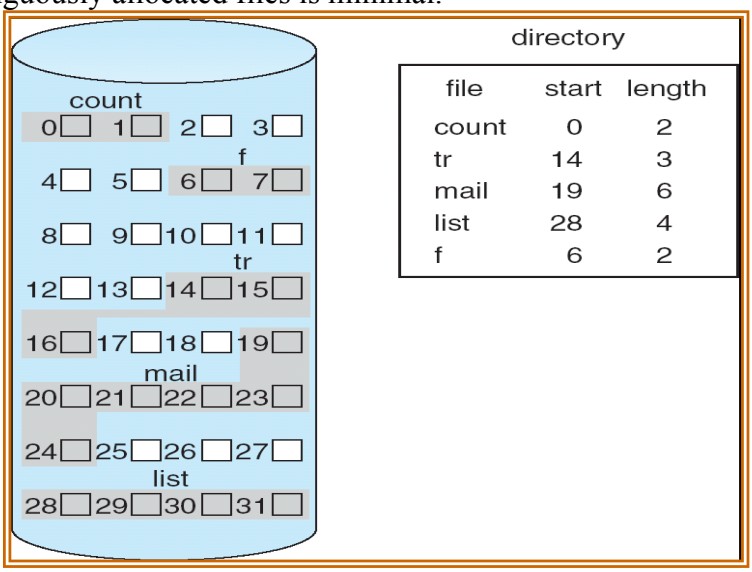


Figure 123: Contiguous Allocation

- 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.

5.4.2. 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.

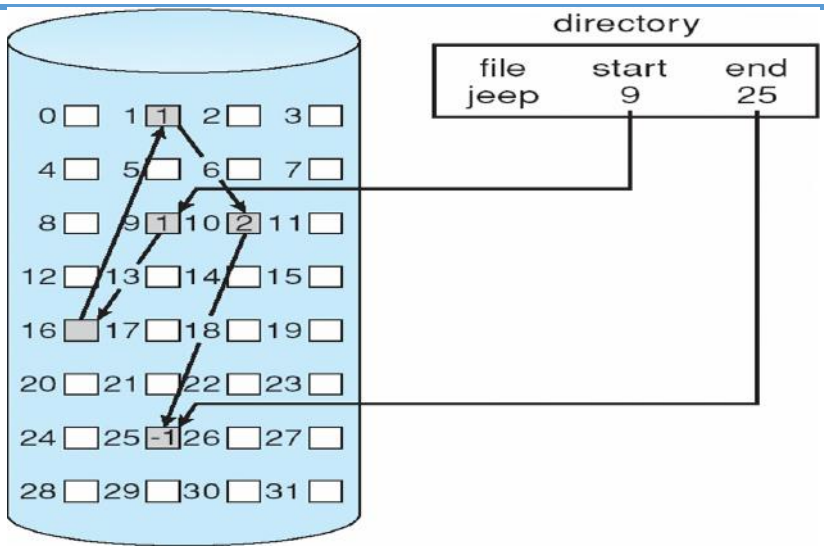


Figure 124: Linked Allocation

- The major problem is that it can be used effectively for only sequential access files. To find the i^{th} block of a file we must start at the beginning of that file, and follow the pointers until we get to the i^{th} 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 define a cluster as 4 blocks and operate 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 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-of file value. An illustrative example is the FAT structure of for a file consisting of disk blocks 217, 618, and 339.

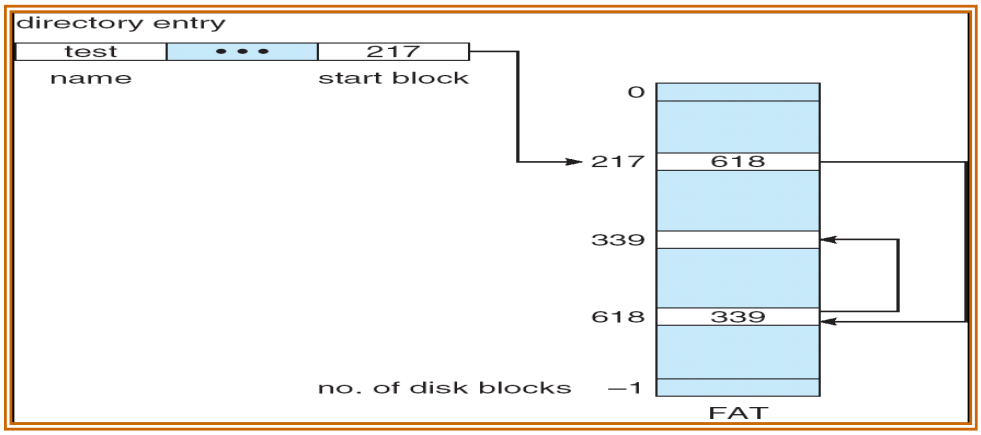


Figure 125: File Allocation Table

5.4.3. 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 the 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.

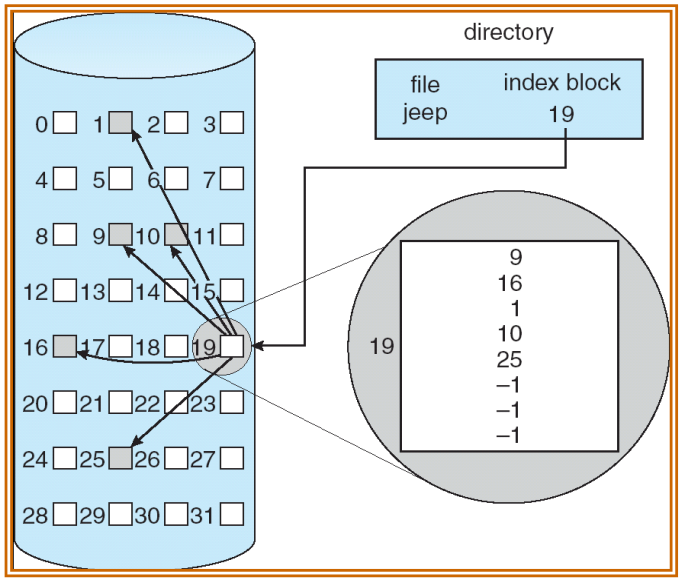


Figure 126: Index Allocation

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.

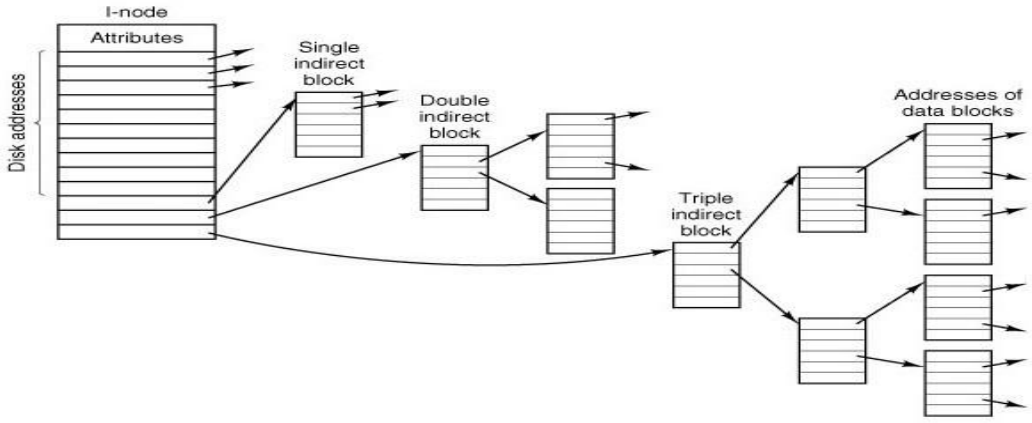
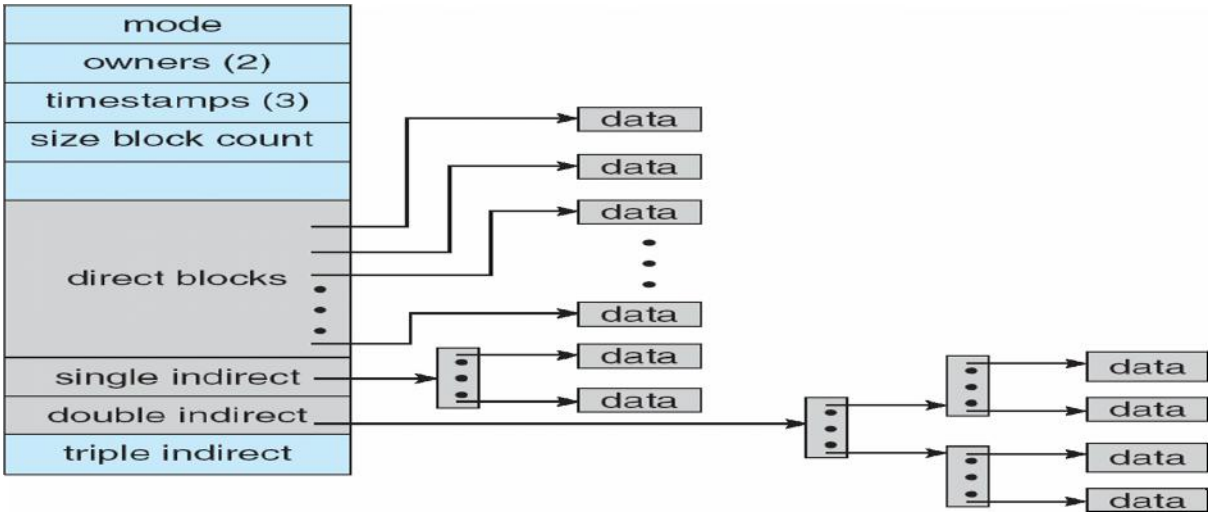


Figure 127: An i-node with three levels of indirect blocks

5.4.4. Combined Scheme: UNIX UFS



More index blocks than can be addressed with 32-bit file pointer

Figure 128: Combined Scheme: UNIX UFS

Performance

- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

5.5. 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.

5.5.1. 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 …..
- 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

5.5.2. 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.

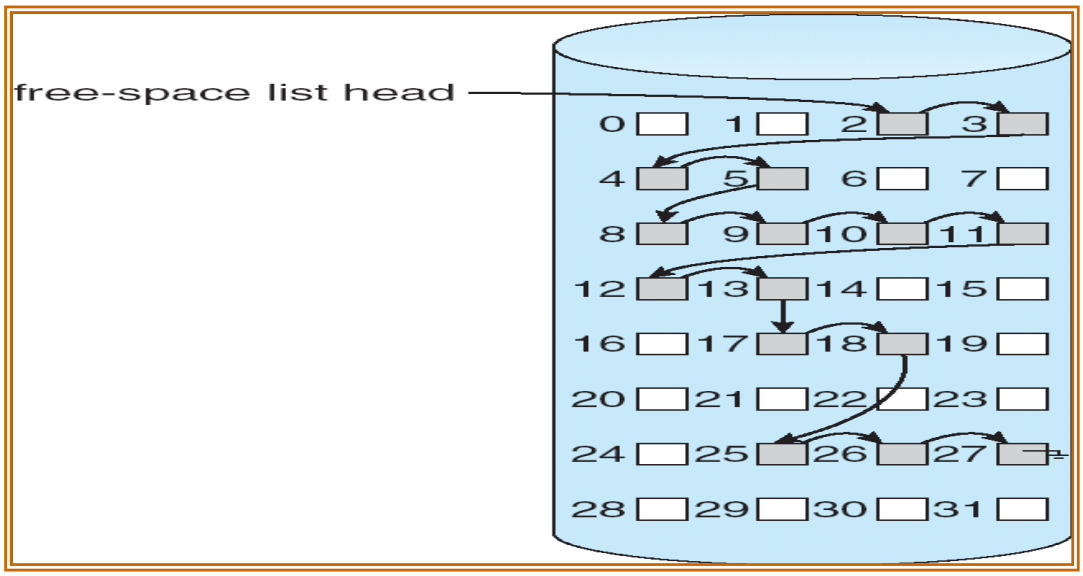


Figure 129: Linked List scheme

5.5.3. 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.

5.5.4. 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.

5.6. Efficiency and Performance

- Efficiency dependent on:
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry
  - Pre-allocation or as-needed allocation of metadata structures
  - Fixed-size or varying-size data structures
- Performance
  - Keeping data and metadata close together
  - disk cache – separate section of main memory for frequently used blocks
  - **Synchronous** writes sometimes requested by apps or needed by OS
    - No buffering / caching – writes must hit disk before acknowledgement
    - **Asynchronous** writes more common, buffer-able, faster
  - free-behind and read-ahead – techniques to optimize sequential access
  - Reads frequently slower than writes
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk

5.7. Page Cache

- A page cache caches pages rather than disk blocks using virtual memory techniques
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache

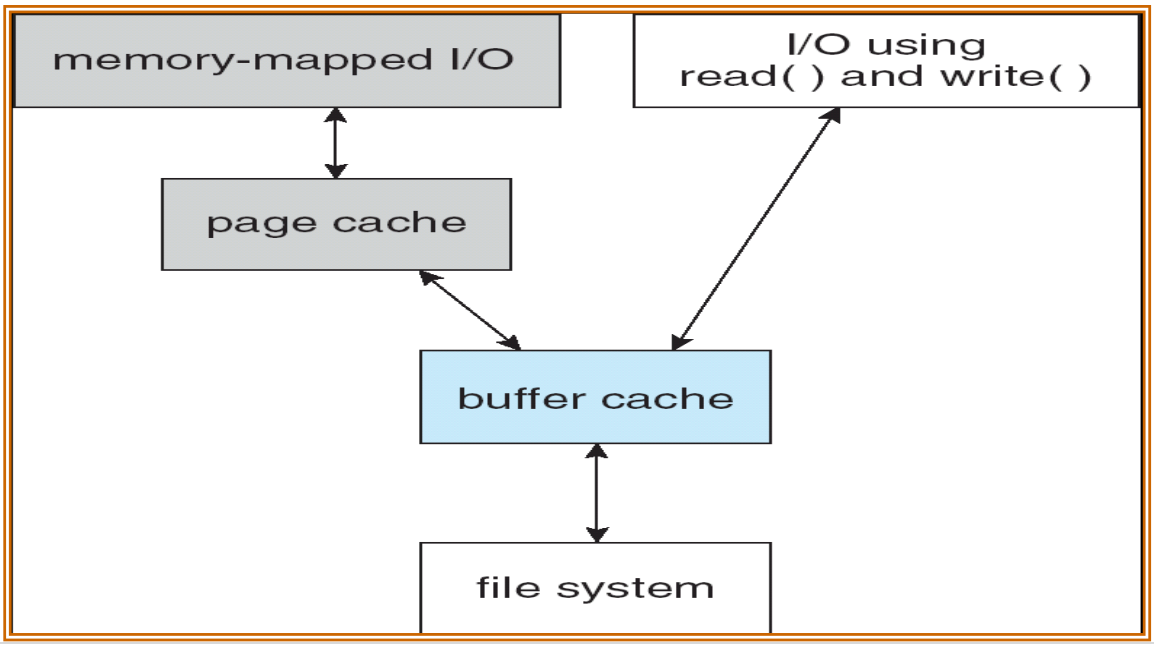


Figure 130: I/O without unified buffer

A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O

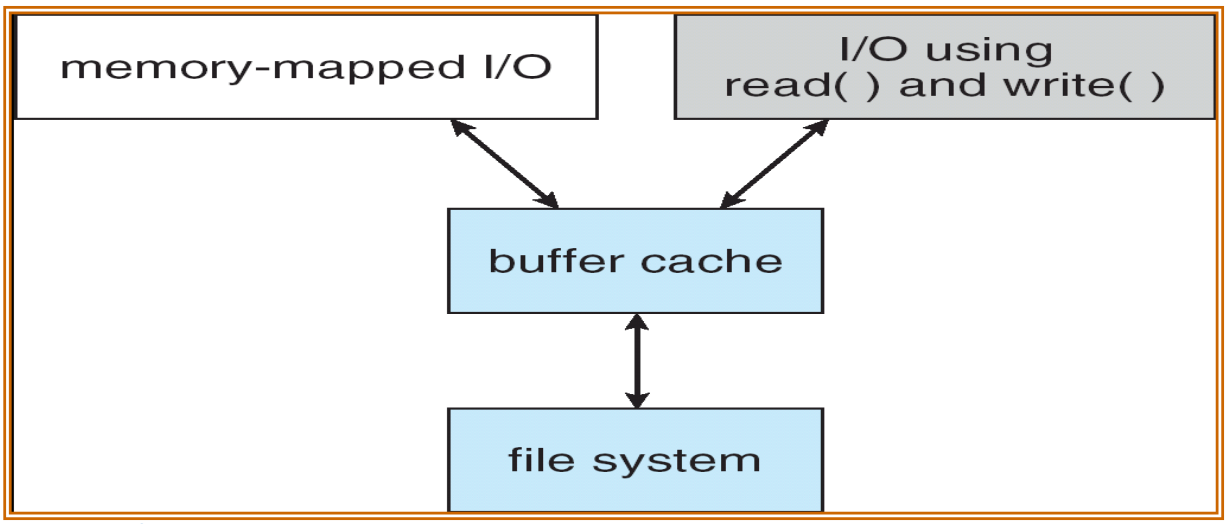


Figure 131: I/O using unified buffer

5.8. Recovery

- **Consistency checking** – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
  - Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

5.8.1. Log Structured File Systems

- Log structured (or journaling) file systems record each update to the file system as a transaction
- All transactions are written to a log
  - A transaction is considered committed once it is written to the log
  - However, the file system may not yet be updated
- The transactions in the log are asynchronously written to the file system
  - When the file system is modified, the transaction is removed from the log
- If the file system crashes, all remaining transactions in the log must still be performed
- Faster recovery from crash, removes chance of inconsistency of metadata

5.9. Directory Implementation

- Linear list of file names with pointer to the data blocks.
  - simple to program
  - time-consuming to execute
- Hash Table – linear list with hash data structure.
  - decreases directory search time
  - *collisions* – situations where two file names hash to the same location
  - fixed size

5.9.1. Linear list

- Simplest method of implementing a directory
- Uses a linear list of file names with pointer to the data blocks
- A linear list or directory entries requires a linear search to find a particular entry.
- This method is simple to program but is time consuming to execute
- To create a new file we must first search the directory to be sure that no existing file has the same name. If no any filename exists then we add a new entry at the end of the directory.
- To delete a file we search the directory for the named file then release the space allocated to it
- The real disadvantage of a linear list of directory entries is the linear search to find a file

5.9.2. Hash table

- In this method linear list stores the directory entries but a hash data structure is also used
- This method decreases the directory search time.
- Some provisions must be made for collisions (situation where two filenames hash to the same location)
- Major difficulties with hash table are:
- Fixed size of the hash table and
- Dependence of the hash function on the size of hash table.

5.10. Backup and restore

- After back up we can recover the loss of an individual file by restoring the data from backup
- When to do back up?
- A typical schedule is as follows:
  - Day 1: copy to a backup medium all files from the disk– also called full backup
  - Day 2: copy to another medium all files changed since day 1– an incremental backup
  - Day 3: copy to another medium all files changed since day 2
  - Day N: copy to another medium all files since day N-1. then go back to day 1

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.
- It is useful to create directories to allow files to be organized.

- A single-level directory in a multiuser system causes naming problems, since each must have a unique name.
- A two-level directory solves this problem by creating a separate directory for each user's files.
- The directory lists the files by name and includes the file's location on the disk, length, type, owner, time of creation, time of last use, and so on.
- The natural generalization of a two-level directory is a tree structured directory.
- A tree-structured directory allows a user to create subdirectories to organize files.
- Acyclic-graph directory structures enable users to share subdirectories and files but complicate searching and deletion.
- A general graph structure allows complete flexibility in the sharing of files and directories but sometimes requires garbage collection to recover unused disk space.
- File systems may be mounted into the system's naming structures to make them available. The naming scheme varies by operating system.
- Once mounted, the files within the volume are available for use.
- File sharing depends on the semantics provided by the system.
- Files may have multiple readers, multiple writers, or limits on sharing.
- Distributed file systems allow client hosts to mount volumes or directories from servers, as long as they can access each other across a network.
- Remote file systems present challenges in reliability, performance, and security.
- 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.
- The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently.
- The most common secondary-storage medium is the disk.
- Physical disks may be segmented into partitions to control media use and to allow multiple, possibly varying, file systems on a single spindle.
- These file systems are mounted onto a logical file system architecture to make them available for use.
- File systems are often implemented in a layered or modular structure.
- The lower levels deal with the physical properties of storage devices.
- Upper levels deal with symbolic file names and logical properties of files.
- Intermediate levels map the logical file concepts into physical device properties.
- A VFS layer allows the upper layers to deal with each file-system type uniformly.
- The various files can be allocated space on the disk in three ways: through contiguous, linked, or indexed allocation.
- Contiguous allocation can suffer from external fragmentation.
- Direct access is very inefficient with linked allocation.
- Indexed allocation may require substantial overhead for its index block.
- Contiguous space can be enlarged through extents to increase flexibility and to decrease external fragmentation.
- Indexed allocation can be done in clusters of multiple blocks to increase throughput and to reduce the number of index entries needed.
- Indexing in large clusters is similar to contiguous allocation with extents.
- Free-space allocation methods also influence the efficiency of disk-space use, the performance of the file system, and the reliability of secondary storage.
- The methods used include bit vectors and linked lists.

- Optimizations include grouping, counting, and the FAT, which places the linked list in one contiguous area.
- Directory-management routines must consider efficiency, performance, and reliability.
- A hash table is a commonly used method, as it is fast and efficient.
- Unfortunately, damage to the table or a system crash can result in inconsistency between the directory information and the disk's contents.
- A consistency checker can be used to repair the damage. Operating-system backup tools allow disk data to be copied to tape, enabling the user to recover from data or even disk loss due to hardware failure, operating system bug, or user error.
- Network file systems, such as NFS, use client-server methodology to allow users to access files and directories from remote machines as if they were on local file systems.
- System calls on the client are translated into network protocols and retranslated into file-system operations on the server.
- Networking and multiple-client access create challenges in the areas of data consistency and performance.
- Due to the fundamental role that file systems play in system operation, their performance and reliability are crucial.
- Techniques such as log structures and caching help improve performance, while log structures and RAID improve reliability.

References

1. Silberchatz, P.B. Galvin and G. Gange, "*Operating System Concepts*", Eight Edition, Wiley Publishing; ISBN:0470128720
2. A.S. Tanenbaum, H.Bos: "*Modern Operating Systems*", Fourth Edition; Prentice Hall Press: ISBN-13:9780133591620