Ceng 328 Operating Systems Lecture Notes

Cem Özdoğan

25th May 2004

Contents

1 Introduction **13**

1.1 What is Operating System? 14
1.1.1 Functionalities in OS: 15
1.2 History of operating Systems 15
1.2.1 Mainframe Systems 16
1.2.2 Batch and Multiprogrammed Systems 16
1.2.3 Time Sharing 18
1.2.4 Personnel Computers 20
1.2.5 Parallel Processing Systems 20
1.2.6 Distributed Systems 22
1.2.7 Clustered Systems 23
1.2.8 Real–Time 23
1.2.9 Embedded, Smart-Card and Handheld 24
1.3 Computer Hardware Review 25
1.3.1 Processor 25
1.3.2 Main Memory 27
1.3.3 I/O Modules and Structure 29
1.3.4 System bus 30
1.3.5 Storage Structure and Hierarchy 30
1.3.6 Protection 31
1.4 Interrupts and Traps 34
1.4.1 Accessing OS Services 37
1.5 Operating System Components 38
1.6 Bootstrapping 39
1.7 System Structure 40
1.8 Why Study Operating Systems? 41
1.8.1 Problems in building OS 42

2 Processes and Threads **45**

2.1 Processes 45
2.1.1 The Process Model 46

2.1.2 Context Switch 48
2.1.3 Dispatcher 50
2.1.4 Process Creation 51
2.1.5 Process Termination 53
2.1.6 Process States 54
2.2 Threads 54
2.2.1 The Thread Model 58
2.2.2 Implementing Threads in User Space 58
2.2.3 Implementing Threads in the Kernel 59
2.3 Interprocess Communication 59
2.3.1 Race Conditions 61
2.3.2 Critical Regions 61
2.3.3 Mutual Exclusion with Busy Waiting (Software approach) 62
2.3.4 Sleep and wakeup 64
2.3.5 Semaphores 68
2.3.6 Monitors 71
2.4 Classical IPC Problems 74
2.4.1 The Dining Philosophers Problem (see Fig. 2.17) . . . 74
2.4.2 The Readers and Writers Problem (see Fig. 2.19) . . . 76
2.4.3 The Sleeping Barber Problem (see Fig. 2.20) 77
2.5 Scheduling 79
2.6 Introduction to Scheduling 79
2.7 Scheduling in Batch Systems 82
2.8 Scheduling in Interactive Systems 84
2.9 Policy versus Mechanism 88

3 Deadlock **89**
3.1 Resources 90
3.2 Introduction to Deadlocks 90
3.2.1 Conditions for Deadlock (See Fig. 3.1) 90
3.2.2 Deadlock Modeling (See Fig. 3.2) 92
3.3 The Ostrich Algorithm 93
3.4 Deadlock Detection and Recovery 93
3.5 Deadlock Avoidance 96
3.5.1 Resource Trajectories (See Fig. 3.4) 96
3.5.2 Safe and Unsafe States (See Fig. 3.5) 97
3.5.3 The Banker's Algorithm for Deadlock Avoidance 98
3.6 Deadlock Prevention 100
3.7 Summary of Deadlock strategies 102

4 Memory Management 103
4.1 Basic Memory Management 105
4.1.1 Monoprogramming without Swapping or Paging (see the Fig. 4.3)106
4.1.2 Multiprogramming with Fixed Partitions (see the Fig. 4.4)107
4.1.3 Relocation and Protection (see the Fig. 4.5) 107
4.2 Swapping 108
4.2.1 Memory Management cont. 111
4.3 Virtual Memory 113
4.3.1 Paging (see the Fig. 4.8) 113
4.3.2 Page Tables (see Fig. 4.12) 118
4.3.3 Inverted Page Tables 120
4.3.4 Basic policies 120
4.3.5 Page Replacement Algorithms 122
4.3.6 Page Replacement Cont. 123
4.4 Segmentation 126
4.4.1 Segmentation with Paging 128
4.4.2 Segmentation with Paging: MULTICS 130
4.4.3 Segmentation with Paging: The Intel Pentium (see Fig. 4.18)131

5 INPUT/OUTPUT 133
5.1 Principles of I/O Hardware 133
5.1.1 Device Controllers (see Fig. 5.1) 134
5.1.2 I/O Devices 134
5.1.3 Characteristics (see Table 5.2) and Differences in I/O Devices136
5.1.4 Evolution of the I/O Function (see Fig. 5.2) 137
5.1.5 Memory-Mapped I/O (see Fig. 5.3) 138
5.1.6 Direct Memory Access (DMA) 139
5.1.7 Interrupts Revisited (see Fig. 5.6) 141
5.2 Principles of I/O Software 141
5.2.1 Programmed I/O (see Fig. 5.7a) 141
5.2.2 Interrupt-Driven I/O (see Fig. 5.7b) 142
5.2.3 Direct Memory Access (see Fig. 5.7c) 142
5.3 Operating System Design Issues 143
5.4 I/O Software Layers (see Fig. 5.8) 143
5.4.1 Interrupt Handlers 143
5.4.2 Device Drivers (see Fig. 5.9) 145
5.4.3 Device Independent I/O Software(see Fig. 5.10) 146
5.4.4 User Level Software 149
5.5 Disks (see Fig. 5.13) 150
5.5.1 Disk Hardware 151
5.5.2 Disk Formatting 154

5.5.3 Disk Arm Scheduling Algorithms (see Fig. 5.18) 155
5.5.4 Error–Handling 156

6 File Systems **159**
6.1 Files 159
6.1.1 File Naming 160
6.1.2 File Structure; From OS's perspective 161
6.1.3 File Types 161
6.1.4 File Access 162
6.1.5 File Attributes(see Fig. 6.4) 163
6.1.6 File Operations 164
6.1.7 An Example Program Using File System Calls (see Fig. 6.6) 165
6.1.8 Memory–Mapped Files (see Fig. 6.7) 165
6.1.9 File Organization and Access; Programmer's Perspective 165
6.2 Directories 171
6.2.1 Path Names 173
6.2.2 File Sharing 174
6.3 File System Implementation 176
6.3.1 File System Layout 176
6.3.2 Implementing Files (see Fig. 6.13) 176
6.3.3 Implementing Directories 181
6.3.4 Shared Files (see Fig. 6.17) 182
6.3.5 Disk Space Management (Free space management) . . 182
6.3.6 Other file system implementation issues 184
6.4 UNIX File Management 186
6.5 vita 193

List of Tables

2.1 Race Condition 61

3.1 Summary of Deadlock strategies 102

4.1 Page replacement algorithms 123

5.1 Device I/O Port Locations on PCs (Partial). 134
5.2 Characteristics of I/O Devices 136

List of Figures

1.1 Abstract view 14
1.2 Job Interleaving 18
1.3 Memory Layout; Simple Batch, Multi Programming 19
1.4 SMP architecture, Client-Server 22
1.5 Fetch and Execute 27
1.6 Cache Memory 28
1.7 Top-level Components, Pentium System 30
1.8 Going down the hierarchy 30
1.9 Interrupt 34
1.10 Interrupt Cycle 36
1.11 Interrupt Picture 37
1.12 System Call 38
1.13 OS Architecture 39
1.14 UNIN System initilization 41
1.15 OS Structures, MS-DOS, Unix, IBM VM/370, Chorus 42

2.1 System Calls 46
2.2 A UNIX Process Context 47
2.3 Process Control Block (PCB), Processes from main memory to registers. 49
2.4 CPU Switch From Process to Process 50
2.5 Diagram of Process State 55
2.6 Single and Multithreaded Processes 56
2.7 Threads and Processes 57
2.8 A word processor with three threads, a multithreaded web server 57
2.9 Thread Models; Many-to-One, One-to-One, Many-to-Many . . 59
2.10 Mutual exclusion using critical regions 63
2.11 A proposed solution to the CR problem. (a) Process 0, (b) Process 1 64
2.12 Peterson' solution for achieving mutual exclusion 65
2.13 The producer-consumer problem with a fatal race problem . . 67
2.14 The producer-consumer problem using semaphore 70
2.15 A monitor 72

2.16 The producer-consumer problem with a monitor. 73
2.17 Lunch time in the Philosophy Department. 74
2.18 A solution to the dining philosophers problem. 75
2.19 A solution to the readers and writers problem. 76
2.20 A solution to the sleeping barber problem. 78
2.21 Some goals of the scheduling algorithm under different circumstances. 81
2.22 An example to First-Come First Served. 83
2.23 An example to Shortest Job First. 83
2.24 Example of non-preemptive SJF and example of preemptive SJF. 84
2.25 An example to Round Robin. 85
2.26 An example to Priority-based Scheduling. 86
2.27 Multi-level queue and Multi-level feedback queue (lower). . . . 87

3.1 An example to Deadlock. 91
3.2 Resource Allocation Graphs, in the right one, either P_2 or P_4 could relinquish a res
3.3 An example of how deadlock occurs and how it can be avoided. 94
3.4 Two process resource trajectories. 97
3.5 Demonstration that the state in is safe (upper), and in is not safe (lower). 98

4.1 Allocating Memory. 104
4.2 From source to executable code. 105
4.3 Three simple ways of organizing memory with an operating system and one user pr
4.4 (a) Fixed memory partitions with separate input queues for each partition. (b) Fix
4.5 Address Translation. 109
4.6 Swapping. 109
4.7 (a) A part of memory with five processes and three holes. The tick marks show the
4.8 The relation between virtual address and physical memory addresses is given by a p
4.9 The position and function of the MMU. 115
4.10 Page fault handling by picture. 117
4.11 Memory Caching. 118
4.12 The internal operation of the MMU with 16 4-KB pages. . . . 119
4.13 Page Replacement. 122
4.14 Trashing. 124
4.15 Logical View of Segmentation (left) , User's View of a Program (right).126
4.16 Example of Segmentation 127
4.17 Sharing of Segmentation 128
4.18 Intel 386 Address Translation 131

5.1 A kernel I/O structure. 135
5.2 Evolution of the I/O Function 137
5.3 a) Separate I/O and memory space. b) Memory-mapped I/O. c) Hybrid.138

5.4 a) A single-bus architecture. b) A dual-bus memory architecture.139
5.5 The Process to Perform DMA Transfer. 140
5.6 How interrupts happen. The connections between devices and interrupt controller actually
5.7 a) Programmed I/O. b) Interrupt-Driven I/O. c) Direct Memory Access.142
5.8 Layers of the I/O Software System. 144
5.9 Logical positioning of device drivers. In reality all communications between drivers and dev
5.10 (a) Without a standard driver interface (b) With a standard driver interface.146
5.11 (a) Unbuffered input (b) Buffering in user space (c) *Single buffering* in the kernel followed b
5.12 Networking may involve many copies. 149
5.13 Disk Structure. 151
5.14 Left: (a) Physical geometry of a disk with two zones (b) A possible virtual geometry for thi
5.15 Disk Performance. 152
5.16 Left: Low-level Disk Formatting; A disk sector, Right: An illustration of cylinder skew.154
5.17 a) No interleaving b) Single interleaving c) Double interleaving.155
5.18 From left to right: First-in, First-out (FIFO); Shortest Seek Time First; Elevator Algorithm
5.19 a) A disk track with a bad sector b) Substituting a spare for the bad sector c) Shifting all t

6.1 Some typical file extensions. 160
6.2 Three kinds of files. (a) byte sequence. (b) record sequence. (c) tree.162
6.3 (a) An executable file (b) An archive. 163
6.4 Some possible file attributes. 164
6.5 File operations. 165
6.6 A simple program to copy a file. 166
6.7 Left: (a) Segmented process before mapping files into its address space (b) Process after ma
6.8 Fundamental File Organizations; (a) Pile (b) Sequential (c) Indexed Sequential (d) Indexed
6.9 IBM indexed-sequential access method (ISAM). 170
6.10 (a) Single-Level Directory Systems (b) Two-Level Directory Systems (c) Hierarchical Direct
6.11 Example to (a) Single-Level Directory Systems (b) Two-Level Directory Systems (c) Hierar
6.12 The solid curve (left hand scale) gives data rate of a disk. The dashed curve (right hand sca
6.13 Left: A possible file system layout. Right: File Allocation Table.177
6.14 (a) Contiguous allocation (b) Contiguous allocation with compaction (c) Storing a file as a
6.15 (a) Indexed allocation with block partitions (b) Indexed allocation with variable–length par
6.16 Left: (a) A simple directory containing fixed–sized entries with the disk addresses and attri
6.17 Left: File system containing a shared file. Right: (a) Situation prior to linking (b) After the
6.18 (a) Storing the free list on a linked list (b) A bit map. 184
6.19 Inode contents. 189
6.20 Left: Direct Block. Right: Single Indirect Block 190
6.21 Upper: System V Disk Layout (s5fs). Middle: Layout of an Ext2 Partition. Lower: Layout

Chapter 1

Introduction

Computer systems have two major components:

- *hardware*–electronic,mechanical, optical devices
- *software*–programs.

Without support software, a computer is of little use. With its *software*, however, a computer can store, manipulate, and retrieve information, and can engage in many other activities. Software can be grouped into the following categories:

- *systems software* (operating system & utilities)
- *applications software* (user programs)

Summary,

- Hardware provides basic computing resources (CPU, memory, I/O devices).
- Operating system controls and coordinates the use of the hardware among the various application programs for the various users.
- Application programs define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
- Users (people, machines, other computers).

1.1 What is Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware.

- An operating system (OS) is everything in system that isn't an application or hardware
- An OS provides orderly and controlled allocation and use (i.e., *sharing, optimization of resource utilization*) of the resources (Processor, Memory, I/O devices) by the users (jobs) that compete for them.
- Support programs (typically called daemons) running in the machine that handle higher level services such as mail transport (networking), off-line file system checking (system robustness), web serving (server work), etc.
- Protection and Security
- Provides an abstraction layer over the concrete hardware. Use the computer hardware in an efficient manner (converting hardware into useful form;) "hide" the complexity of the underlying hardware and give the *user* a better view (an abstraction) of the computer for applications by providing:
  - Standard library
    * allow applications to reuse common facilities
    * make different devices look the same
    * provide higher-level abstractions

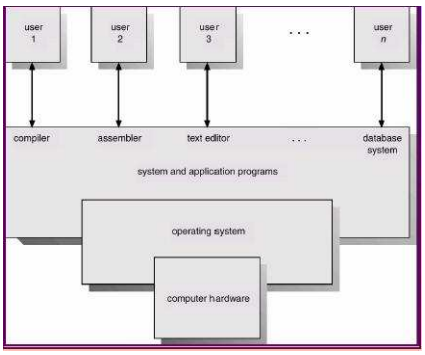


Figure 1.1: Abstract view

    * What are the right abstractions *Challenge*.
  - Resource manager, Resource - "Something valuable" e.g. CPU, memory (RAM), I/O devices (disk). Each program gets **time** with the resource and each program gets **space** on the resource
    * Multiple users/applications can share, why share: (1) devices are expensive, and (2) there is need to share data as well as communicate
    * Protect applications from one another
    * Provide fair and efficient access to resources
    * OS cannot please all the people all the time, but it should please most of the people most of the time, so: What mechanisms? What policies? (e.g.,. which user/process should get priority for printing on a common shared printer?); *Challenges*

1.1.1 Functionalities in OS:

Desired functionalities of OS depend on outside factors like users' & application's "Expectations" and "Technology changes" in Computer Architecture (hardware).
OS must adapt:

- Change abstractions provided to users
- Change algorithms to change these abstractions
- Change low-level implementation to deal with hardware

The current operating systems are driven by such evolutions.

1.2 History of operating Systems

The earliest computers, developed in the 1940s, were programmed in *machine language* and they used front panel switches for input. The programmer was also the operator interacting with the computer directly from the system console (control panel).

- programmers needed to sign-up in advance to use the computer one at a time
- executing a single program (often called a *job*) required substantial time to setup the computer.

- First generation 1945 - 1955, vacuum tubes, plug boards
- Second generation 1955 - 1965, transistors, batch systems
- Third generation 1965 - 1980, ICs and multiprogramming
- Fourth generation 1980 - present, personal computers
- Next generation ??, personal digital assistants (PDA), information appliances

Two distinct phases in history: Expensive computers, Cheap computers

1.2.1 Mainframe Systems

First commercial systems: Enormous, expensive and slow, I/O: Punch cards and line printers.

- Single operator/programmer/user runs and debugs interactively:
  - Standard library with no resource coordination
  - Monitor that is always resident
    - initial control in monitor
    - control transfers to job
    - when job completes control transfers back to monitor
- Inefficient use of hardware: poor *throughput* and poor *utilization*
- *Performance metrics:*
  Throughput: like amount of useful work done per hour
  Utilization: keeping all devices busy
- Mainframe systems started to appear after world war 2.
- They initially executed one program at a time and were known as batch systems.

1.2.2 Batch and Multiprogrammed Systems

Group if jobs submitted to machine together, *Batch*. A job was originally presented to the machine (and its human operator) in the form of a set of cards – these cards held information according to how "punched" out of the cardboard. The operator grouped all of the jobs into various batches with similar characteristics before running them (all the quick jobs might run, then the slower ones, etc.).

- Operator collects job, orders efficiently, runs one at a time
- Amortize setup costs over many jobs
- Keep machine busy while programmer thinks
- User must wait for results until batch collected and submitted

Result: Improved system throughput and utilization, but lost interactivity.
Since the I/O used slow mechanical devices, the CPU was often idle waiting on a card to be read or some result to be printed, etc. One way to minimize this inefficiency was to have a number of jobs available and to switch to running another one to avoid idleness.

- Mechanical I/O devices much slower than CPU
- Overlap I/O with execution by providing pool of ready jobs
- New OS Functionality evolved: Buffering, Direct Memory Access (DMA), interrupt handling

Result: Improves throughput and utilization.
In multiprogrammed systems, a number of programs were resident in memory and the CPU could choose which one to run. One way to choose is to just keep executing the current program until an I/O delay is pending – instead of just waiting, the CPU would move onto the next program ready to be run.

- Keep multiple jobs resident in memory
- OS chooses which job to run
- When job waits for I/O switch to another resident job

Result: Job scheduling policies, Memory management and protection, improves throughput and utilization, still not interactive.

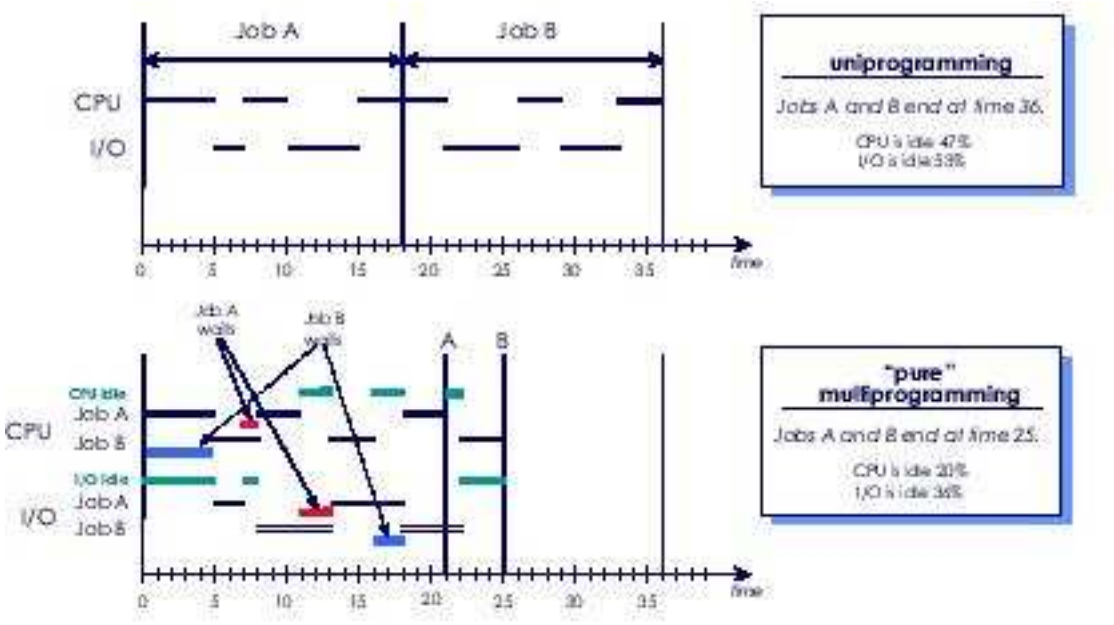


Figure 1.2: Job Interleaving

1.2.3 Time Sharing

While multiprogrammed systems used resources more efficiently i.e. minimized CPU idle time, a user could not interact with a program. By having the CPU switch between jobs at relatively short intervals, we can obtain an interactive system.That is, a system in which a number of users are sharing the CPU (or other critical resource) with a timing interval small enough not to be noticed e.g. no more than 1 second. We say that a time-sharing system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.

- Switch between jobs so frequently that get appearance of dedicated machines for each user/process.
- New OS Functionality: More complex job scheduling, memory management, concurrency control and synchronization
- Users easily submit jobs and get immediate feedbacks

OS Features Needed for Multiprogramming:

- I/O routines supplied by system

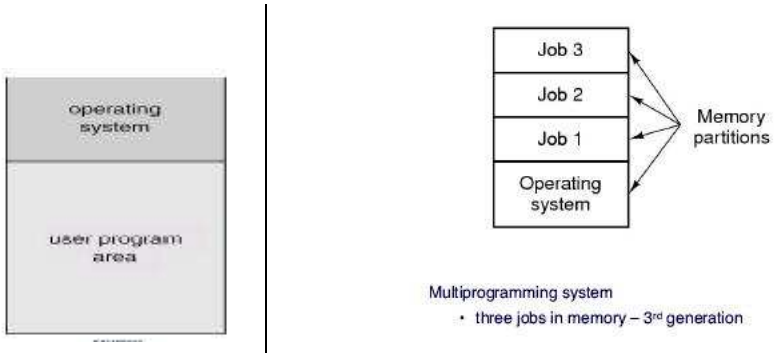


Figure 1.3: Memory Layout; Simple Batch, Multi Programming

- Memory management system must allocate memory to several jobs
- CPU scheduling system must choose among several jobs ready to run
- Allocation of devices

Time-Sharing Systems Interactive Computing:

- The CPU multiplexed among several jobs in memory and on disk (CPU allocated only to jobs in memory). The CPU switches to the next job that can be run whenever the current job enters a wait state or after the current job has used a standard unit of time.When viewed over a relatively long time frame, we obtain the appearance that the CPU is simultaneously running multiple programs.
- Job swapped in and out of memory to disk. If the time-sharing computer does not have enough semiconductor memory installed to hold all of the desired programs, then a backing store must be used to temporarily hold the contents relating to some programs when other programs are present in semiconductor memory. In effect, we are now "memory sharing" between competing users (programs). This idea leads to a mechanism called virtual memory.
- On-line communication between user and system provided; when OS finishes execution of a command, it awaits next "control statement" from user.

The sensible sharing of resources such as CPU time and memory must be handled by the operating system, which is just another program running on the computer. For this control program to always be in control, we require

that it never be blocked from running. The operating system, which might in fact be organized like a small number of cooperating programs, will lock itself into memory and then control CPU allocation priority in order that it never be blocked from running.

1.2.4 Personnel Computers

Single-user, dedicated.

- I/O devices keyboards, mice, screens, printers
- User convenience and responsiveness
- Can adopt technology developed for larger system
- Previously thought,
  - individuals have sole use of computer, do not need advanced CPU utilization, protection features
  - still true? See next list of operating systems...
- May run several different types of OS (Windows, Mac OS X, UNIX, Linux)
- Operating systems such as FreeBSD, NetBSD, Mac OS-X and Linux offer multitasking and virtual memory on PC hardware.
- The *BSD world has influenced the world of computing through networking advances, virtual file storage systems, dynamically self-optimizing resource allocation schemes, etc.
- While all *BSD systems have a family history derived from the original non-networking UNIX, Linux is mostly a “work-alike” re-implementation and can be traced back to MINIX which was developed by Tanenbaum for operating system teaching.

1.2.5 Parallel Processing Systems

Traditional multiprocessor system (share a common bus, clock, and memory), tightly-coupled; multiprocessing. The desire for increased throughput has led to system designs in which multiple streams of processing occurs in parallel.

- *Tightly coupled system* processors share memory and a clock; communication usually takes place through the shared memory. For a system with **n** processors that is to run **n** or more separate programs, the speedup may approach **n**. It will not reach **n** because there will be some contention for access to shared elements such as the memory system.

- Advantages of parallelism:

  - higher throughput and better fault tolerance
  - Economical (?)
  - Increased availability ($\neq$ reliability)

- *Symmetric multiprocessing (SMP)*, a symmetric multi-processor system shares the execution of the operating system amongst all of the processors – it is usually multi-threaded and contains no block structures. The CPUs are equal i.e. we say that they are all peers.

  - Each processor runs identical copy of OS
  - Many processes at once without performance loss
  - Most modern operating systems support SMP

- *Asymmetric multiprocessing*, an asymmetric multi-processor system contains a single CPU called the master that carefully controls access to single threaded sections of the kernel – this processor controls the activities of the slave processors. Asymmetric MP systems are less efficient.

  - Each processor is assigned specific task; master processor schedules, allocates work to slave processors.
  - Mostly for specialized high-end computation

- What can we say about an n-processor system that has m $<$ n application programs to run? Unless some of the application programs can support multiple threadsÂ of simultaneous execution, then the speedup may only approach m (since n - m processors are idle).

- As programmers, you will learn about thread models (and support libraries) that allow you to develop multi-threaded applications. Important applications of multi-threading include database servers, and users of databases will be aware that aspects of simultaneous access can require special care.

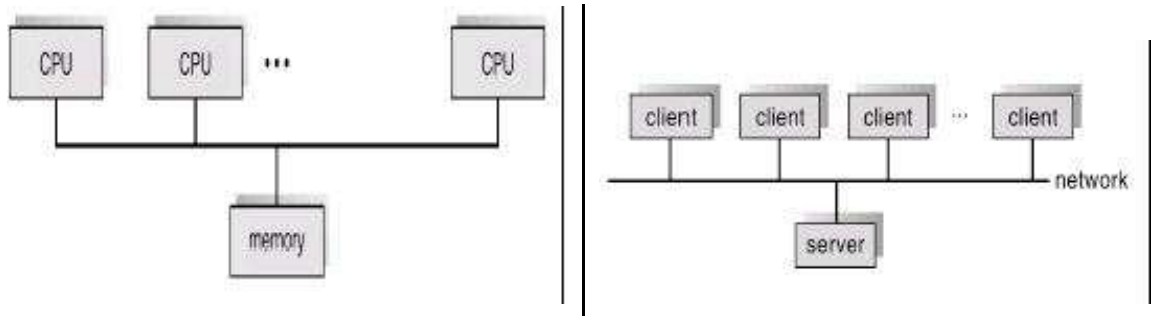


Figure 1.4: SMP architecture, Client-Server

- In specialized application areas such as high speed digital signal processing (DSP) e.g. radar processing, digital mobile base station processing, etc., hybrid parallel systems may be assembled with individual CPUs having significant private memory plus a connection onto a common shared memory.
- While these tightly coupled systems require specialized hardware support in order that the CPUs can share the common memory system, another approach is to use a network to join together more conventional systems into what is termed a distributed system.

1.2.6 Distributed Systems

Multicomputers (do not share memory and clock); loosely-coupled.

- Networked computers
- Require networking infrastructure
- Local area networks (LAN) or Wide area networks (WAN)
- May be either client-server or peer-to-peer systems
  - Client and server roles may vary e.g. X terminal is a windows *server*; runs on machine you think of as *client*
  - Client-Server Systems are a common form of distributed system in which the client system and server system are not similar.
  - An example of a client-server system is the file server on campus. Here, a central server provides file access to authenticated users at client machines.

  - Peer-to-Peer Systems are another form of distributed system in which the participating computer systems are similar.

- Students wishing to experiment with programming at this parallel level can investigate the PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) libraries that are widely available.

1.2.7 Clustered Systems

- 2 or more systems share resources
- Provides high availability
- *Asymmetric clustering*: one server runs application, rest stand by.
- *Symmetric clustering*: all N run application

1.2.8 Real–Time

Deadline (time critical) requirements. A real-time system is required to produce a result within a non-negotiable time period.

- Common uses:
  - control device in dedicated application, e.g., control scientific experiment, medical imaging, industrial control, space shuttle control systems, anti-lock automotive brake systems, banking systems, etc.
  - some display systems
- Real-Time systems may be either;
  - *hard* (must react in time), the real-time system absolutely must complete critical tasks within a guaranteed time.
    - Secondary storage limited or absent, data stored in short term memory, or read-only memory (ROM)
    - Conflicts with time-sharing systems, not supported by general-purpose operating systems.
  - *soft* real-time (deal with failure to react in time), the real-time system can satisfy its performance criteria by running any critical task at a higher priority (of CPU access).
    - Limited utility in industrial control of robotics

  * Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.
  * In some instances, off-the-shelf operating systems such as Linux or *BSD may be modified to support soft real-time operation. An alternative is for the Linux or *BSD operating system to be run as a task within some other (less conventional) real-time operating system.
  * An example of soft real-time service is a multi-media server delivering audio or video – if it fails, no loss of life (other than social life) occurs.

1.2.9 Embedded, Smart-Card and Handheld

- Embedded systems are the most common. They typically run real-time operating systems with custom I/O designed for specific tasks.
- For example, a microwave oven contains a microprocessor chip with built in peripherals such as timers and I/O lines so that cooking may be controlled and keypads and LCD modules handled.
- Personal Digital Assistants (PDAs)
- Cellular telephones, Cameras? ...
- Smart-card and digital mobile telephones also run custom real-time operating systems. At least two "standards" exists – one is JAVA based. The computation load from handling encryption means that the designer has an interesting problem given limited resources of electrical power, memory, and CPU capacity.
- Hand-held systems must also deal with limited resources although their screens have recently become more substantial.
- Issues:
  - Limited memory
  - Variety of interconnect standards
  - Slow processors
  - Small screens
- Their current evolution may be towards a form of "cut back" PC. Sounds like a PC in 1985...

1.3 Computer Hardware Review

Understanding operating systems requires some basic understanding of computer systems.

1.3.1 Processor

- Processor
  - Fetches instructions from memory, decodes and executes them
  - Set of instructions is processor specific
  - Instructions include:
    - load value from memory into register
    - combine operands from registers or memory
    - branch
  - All CPU's have registers to store
    - key variables and temporary results
    - information related to control program execution
- Processor Registers
  - Data and address registers
    - Hold operands of most native machine instructions
    - Enable programmer to minimize main-memory references by optimizing register use
    - user-visible
  - Control and status registers
    - Used by processor to control operating of the processor
    - Used by operating-system routines to control the execution of programs
    - Sometimes not accessible by user (architecture dependent)
- User–Visible Registers
  - May be referenced by machine language instructions
  - Available to all programs - application programs and system programs
  - Types of registers

        * Data
        * Address
            · Index
            · Segment pointer
            · Stack pointer
        * Many architectures do not distinguish different types

- Control and Status Registers
    - Program Counter (PC), Contains the address of an instruction to be fetched
    - Instruction Register (IR), Contains the instruction most recently fetched
    - Processor Status Word (PSW)
        * condition codes
        * interrupt enable/disable
        * supervisor/user mode
    - Condition Codes or Flags
        * Bits set by the processor hardware as a result of operations
        * Can be accessed by a program but not altered
        * Examples: positive/negative result, zero, overflow

- Instruction Fetch and Execute
    - Program counter (PC) holds address of the instruction to be fetched next
    - The processor fetches the instruction from memory
    - Program counter is incremented after each fetch
    - Overlapped on modern architectures (pipelining)

- Instruction Register
    - Fetched instruction is placed in the instruction register
    - Types of instructions
        * Processor-memory, transfer data between processor and memory
        * Processor-I/O, data transferred to or from a peripheral device
        * Data processing, arithmetic or logic operation on data
        * Control, alter sequence of execution

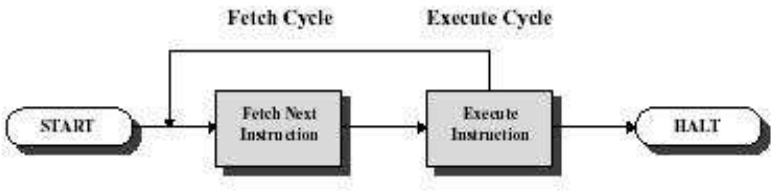


Figure 1.5: Fetch and Execute

1.3.2 Main Memory

- Referred to as real memory or primary memory
- volatile, because its contents are lost when the power is removed
- Should be, fast, abundant, cheap, Unfortunately, that's not the reality..., Solution: combination of fast & expensive and slow & cheap memory
- Program instructions and the data used by programs being executed must reside in high speed semiconductor memory called random access memory (RAM) in order to obtain high speed operation. We say random access because the CPU can access any byte of storage in any order.

Disk Cache

- A portion of main memory used as a buffer to temporarily to hold data for the disk
- Disk writes are clustered
- Some data written out may be referenced again. The data are retrieved rapidly from the software cache instead of slowly from disk
- Mostly transparent to operating system

Cache Memory

- Contain a small amount of very fast storage which holds a subset of the data held in the main memory
- Processor first checks cache

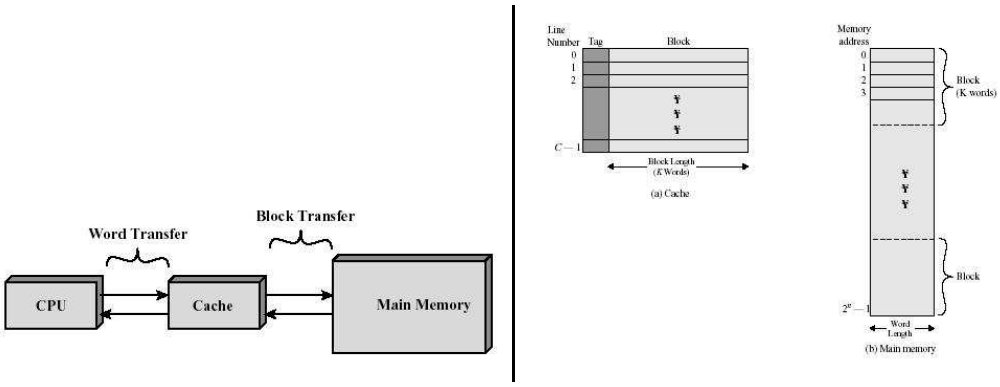


Figure 1.6: Cache Memory

- If not found in cache, the block of memory containing the needed information is moved to the cache replacing some other data

Cache Design

- Cache size, small caches have a significant impact on performance
- Line size (block size), the unit of data exchanged between cache and main memory
- hit means the information was found in the cache
- larger line size ⇒ higher hit rate
- until probability of using newly fetched data becomes less than the probability of reusing data that has been moved out of cache

Mapping function, determines which cache location the data will occupy.
Replacement algorithm

- determines which line to replace
- Least-Recently-Used (LRU) algorithm

Write policy,

- When the memory write operation takes place
- Can occur every time line is updated (write-through policy)
- Can occur only when line is replaced (write-back policy)
  - Minimizes memory operations
  - Leaves memory in an obsolete state

1.3.3 I/O Modules and Structure

- secondary memory devices
- communications equipment
- terminals

CPU much faster than I/O devices

- waiting for I/O operation to finish is inefficient
- not feasible for mouse, keyboard
- I/O module sends an interrupt to CPU to signal completion
- Interrupts normal sequence of execution
- I/O requests can be handled synchronously or asynchronously.
  - In a synchronous system, a program makes the appropriate operating system call and, as the CPU is now executing operating system code, the original program's execution is halted i.e. it waits.
  - In an asynchronous system, a program makes its request via the operating system call and then its execution resumes. It will most likely not have had its request serviced yet!
  - The advantage of having an asynchronous mechanism available is that the programmer is free to organize other CPU activity while the I/O request is handled.
- Software that communicates with controller is called device driver
- Most drivers run in kernel mode
- To put new driver into kernel, system may have to
  - be relinked
  - be rebooted
  - dynamically load new driver
- we must have an event driven I/O system handling all of the pending I/O requests (maybe these are triggered when data arrives, or a peripheral device such as a CD drive indicates it is ready etc.).
- Requests that the operating system has not yet been able to service might mean that the program is currently "sleeping" or "waiting".

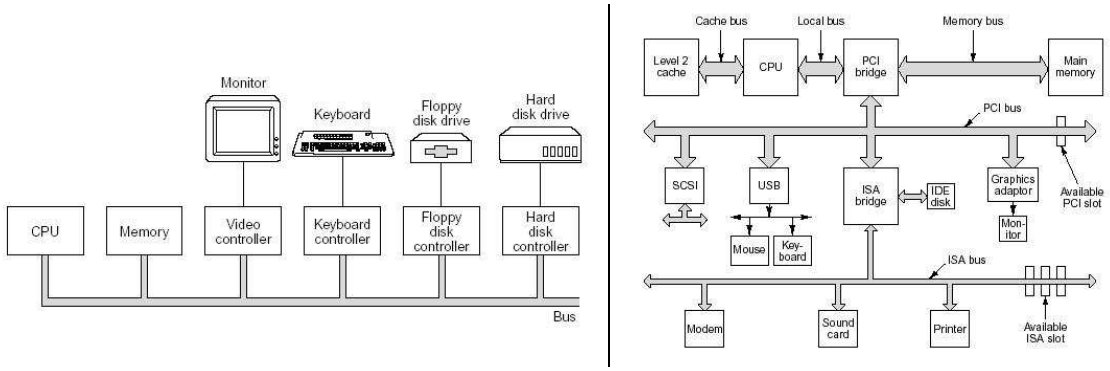


Figure 1.7: Top-level Components, Pentium System

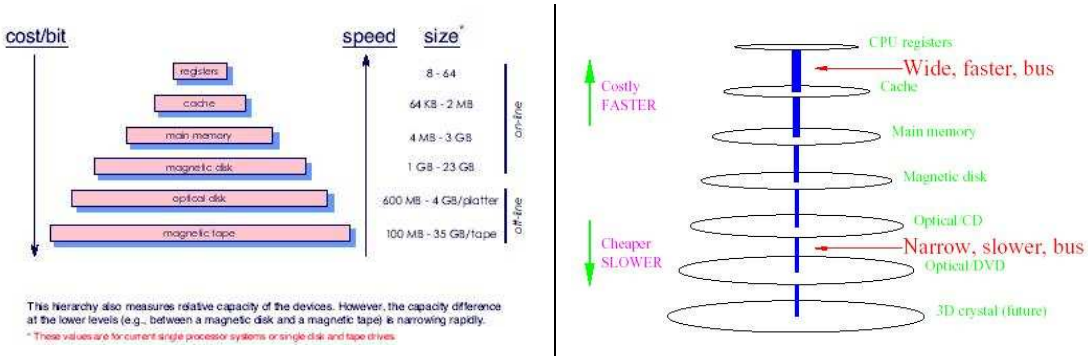


Figure 1.8: Going down the hierarchy

1.3.4 System bus

Communication among processors, memory, and I/O modules.

- A system bus would link the CPU and memory – this structure would involve a pathway along which data could travel (usually 32-bits side-by-side i.e. in bit-wise parallel), a pathway along which the address specifying a particular desired memory location could travel, and a few other lines which would tell the memory whether to store (write) or retrieve (read) data in an access.
- As any I/O device must pass data between the computer and the outside world, it will also be attached to the memory system and the CPU via the system bus.

1.3.5 Storage Structure and Hierarchy

- Decreasing cost per bit

- Increasing capacity
- Increasing access time
- Decreasing frequency of access of the memory by the processor
- Fully electronic memory systems are the fastest and most expensive, hence must be used in cost effective ways. This memory system is called main memory or primary memory.
- A source of cheaper-per-byte and non-volatile storage is provided by magnetic disk. However, the computer does not have direct random access to any byte at any time on the disk – the magnetic discs in the drive are rotating and magnetic heads move in and out in order to access any part of the surface area on the disc that holds data. This means access usually involves a disc rotation delay and also a head positioning delay.
- Other common forms of non-volatile secondary storage include: optical CD drives (CD-R write-once or CD-RW read-write), recent flash memory chips in very small modules that can be inserted into laptop card interfaces or can be used for data logging.
- Stages such as the CPU registers and cache are typically located within the CPU chip so distances are very short and busses can be made very very wide (e.g. 128-bits), yielding very fast speeds.
- Future storage technology includes 3-dimensional crystal structures which allow optical access to a dense 3-dimensional storage facility.

1.3.6 Protection

- Single Tasking System
  - Only one program can perform at a time
  - Simple to implement, Only one process attattempting use resources
  - Few security risks
  - Poor utilization of the CPU and other resources
  - i.e., MS-DOS
- Multi Tasking System

  - Very complex
  - Serious security issues, how to protect one program from another sharing the same memory
  - Much higher utilization of system resources
  - i.e., Unix, Windows NT

- OS must protect itself from users -reserved memory only accessible by OS. The operating system is responsible for allocating access to memory space and CPU time and peripherals etc., and it will control dedicated hardware facilities to help it enforce whatever resource allocation policies are in force:

  - The memory controller, unremarked when it appeared in the basic computer ororganizationis under operating system control to detect and prevent ununauthorizedccess
  - A timer will also be under operating system control to manage CPU time allocation to programs competing for resources

- OS may protect users from another user. A fundamental requirement of multiple users of a shared computer system is that they do not interfere with each other. This gives rise to the need for separation of the programs in terms of their resource use and access;

  - If one program attempts to access main memory allocated to some other program, the access should be denied and an exception raised
  - If one program attempts to gain a larger proportion of the shared CPU time, this should be prevented

- One approach to implementing resource allocation is to have at least two modes of CPU operation, where one mode called the supervisory mode has its code kept in a reserved memory region, and to limit execution of special resource allocation instructions to only the program executing in the supervisory mode.

- Modes of operation

  - supervisor (protected, kernel) mode: *all* (basic and privileged) instructions available.
    * all hardware and memory available
    * mode the OS runs in

    - never let the user run in supervisory mode
  - user mode: a *subset* (basic only) of instructions.
    - limited set of hardware and memory available
    - mode all user programs run in
    - I/O protection, all I/O operations are privileged
    - Memory protection, base/limit registers (in early systems), memory management unit, MMU (in modern systems)
    - CPU control, timer (alarm clock), context switch
  - All I/O instructions are restricted to supervisory mode – so user programs can only access I/O by sending a request to the (controlling) operating system
  - All instructions controlling the memory management unit are restricted to supervisory mode – so user programs can only access the memory that the operating system has allocated
  - All instructions controlling the timer (or real-time clock) are restricted to supervisory mode – so user programs can only read the time of day, and can only have as much CPU time as the operating system allocates
  - All interrupt vector table entries, which are specific to each task or program that can run, must be configured (initially at least) by the (controlling) operating system

- One of the early advantages of UNIX operating systems was that a well defined set of system calls was developed to allow a programmer to request access to system resources.

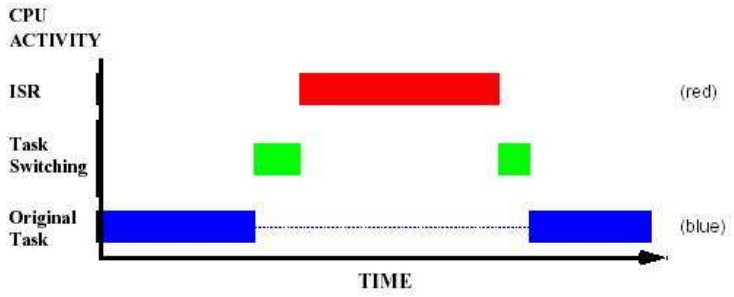


Figure 1.9: Interrupt

1.4 Interrupts and Traps

- Consider the case when data is to be input from the outside world. One approach is to execute a small code fragment to manage the transfer of data in from the outside world e.g. a peripheral. But when should this code fragment be run?
  - In a polling system, the computer periodically executes (or polls) the peripheral device of interest and inputs data when it is available. This means data can only be input if the peripheral device is polled.
  - In an interrupt driven system, the peripheral triggers the execution of the previously mentioned code fragment when it has data ready. We say interrupt because the handling of this data transfer interrupts normal program execution. The original task (blue) has execution interrupted while a task switching occurs (green) and then the interrupt service routine runs (red) to do I/O. Later, the original task resumes.
  - As the system has more than one peripheral, it will need more than one interrupt service routine available. Generally, a separate interrupt service routine (ISR) is provided for each peripheral that needs to trigger some CPU activity so an array of function pointers holds the start address of each of the ISRs provided i.e. **void** $(*isr\_vectors[])()$.
  - This trigger mechanism allows a computer response to an external event – what about internal events? Most computers are also able to trigger special event handling in software by executing a special instruction called software interrupt or trap.

- The operating system gets the control of the CPU (which may be busy waiting for an event or be in a busy loop) when either an *external* or an *internal* event (or an exception) occurs.
  - external events
    - Character typed at console
    - Completion of an I/O operation (controller is ready to do more work)
    - Timer: to make sure operating system eventually gets control.
    - Hardware failure
    - An *interrupt* is the notification of an (external) event that occurs in a way that is *asynchronous* with the current activity of the processor. Exact occurrence time of an interrupt is not known and it is not predictable
  - internal events
    - System call
    - Error item (e.g., arithmetic overflow, division by zero, illegal instruction, addressing violation)
    - Page fault, reference outside user's memory space
    - A *trap* is the notification of an (internal) event that occurs while a program is executing, therefore is *synchronous* with the current activity of the processor. Traps are immediate and are usually predictable since they occur while executing (or as a result of) a machine instruction.
- Interrupt Cycle
  - Fetch next instruction
  - Execute instruction
  - Check for interrupt
  - If no interrupts, fetch the next instruction
  - If an interrupt is pending, divert to the interrupt handler
- Systems that generate interrupts have different priorities for various interrupts; i.e., when two interrupts occur simultaneously, one is serviced "before" the other.

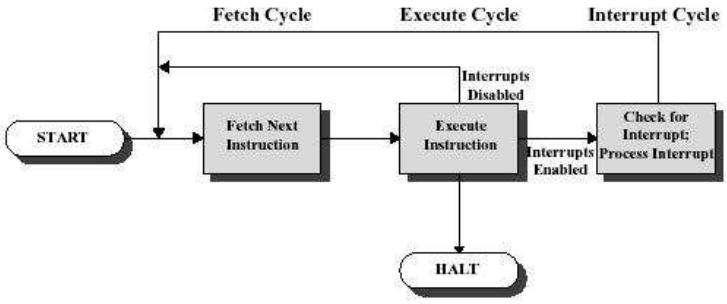


Figure 1.10: Interrupt Cycle

  - When a new "higher priority" interrupt occurs while lesser interrupt is being serviced, the current handler is "suspended" until the new interrupt is processed. This is called the "*nesting of interrupts*."
  - When interruption of an interrupt handler is undesirable, other interrupts can be "*masked*" (inhibited) temporarily

- Interrupt handling by "words". When the CPU receives an interrupt, it is *forced* to a different context (kernel's) and the following occur:
  - The current state of the CPU (PSW) is saved in some specific location
  - The interrupt information is stored in another specified location
  - The CPU resumes execution at some other specific location–the interrupt service routine
  - After servicing the interrupt, the execution resumes at the saved point of the interrupted program
  - Although the details of the above differ from one machine to another, the basic idea remains the same: *the CPU suspends its (current) execution and services the interrupt.*

- Modern languages such as C++ and JAVA allow the programmer to write their own exception handlers. You are writing a special function that can return no result (since it is not so much "called" as "triggered"); and the computing environment is allowing you to store the start address of your exception handler in the array $isr\_vectors[]$.

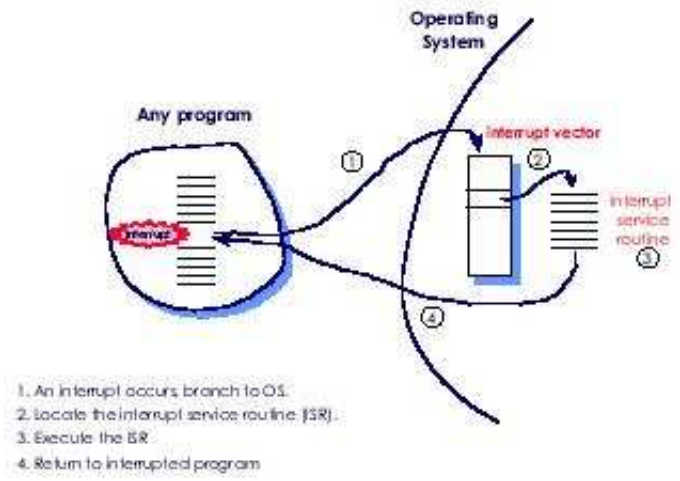


Figure 1.11: Interrupt Picture

- As the computer designers have total control over which event handlers can be accessed by users and which ones are reserved for their use, the exception handling mechanism is also a good way to allow a user program to make a request for a resource from the operating system. We will come across special operations such as a system call or monitor call which are implemented by the exception handling mechanism and provide controlled access to system resources.

1.4.1 Accessing OS Services

- The mechanism used to provide access to OS services (i.e., enter the operating system and perform a "privileged operation") is commonly known as a *system call*. The (only) difference between a "procedure call" and a "system call" is that a system call changes the execution mode of the CPU (to *supervisor mode*) whereas a procedure call does not.

- *System call interface*: A set of functions that are called by (user) programs to perform specific tasks. System call groups:
  - Process control, fork(), exec(), wait(), abort()
  - File manipulation, chmod(), link(), stat(), creat()
  - Device manipulation, open(), close(), ioctl(), select()
  - Information maintenance, time(), acct(), gettimeofday()

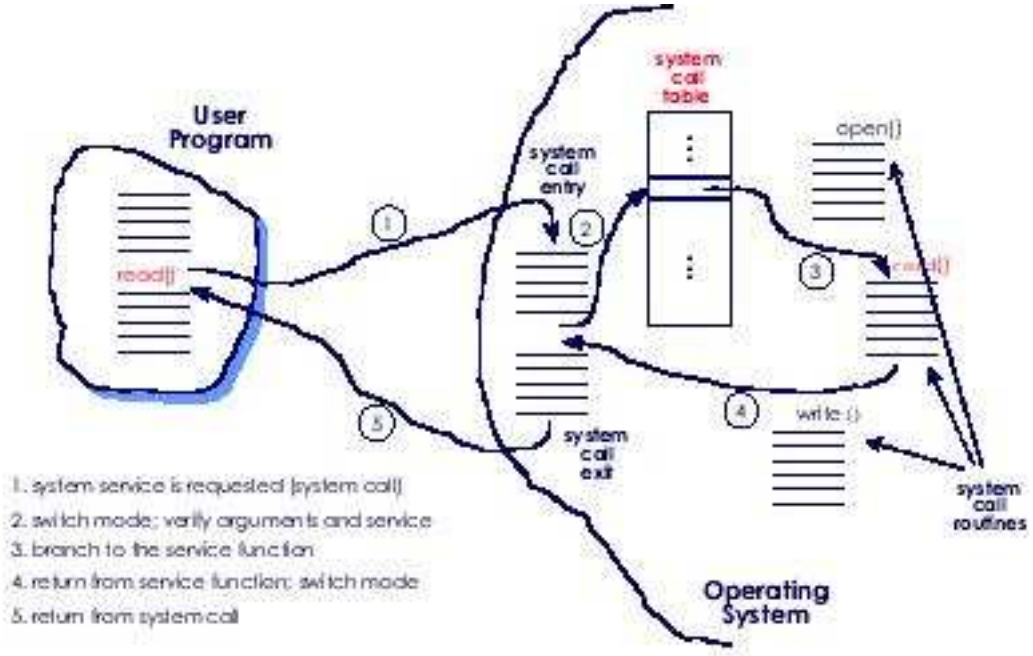


Figure 1.12: System Call

– Communications, socket(), accept(), send(), recv()

1.5 Operating System Components

An operating system generally consists of the following components:

- Process management
- (Disk) storage management
- Memory management
- I/O (device) management
- File systems
- Networking
- Protection
- User Interface

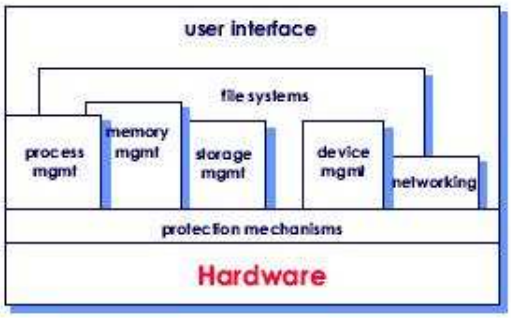


Figure 1.13: OS Architecture

1.6 Bootstrapping

- The process of initializing the computer and loading the operating system is known as *bootstrapping*. This usually occurs when the computer is powered-up or reset.

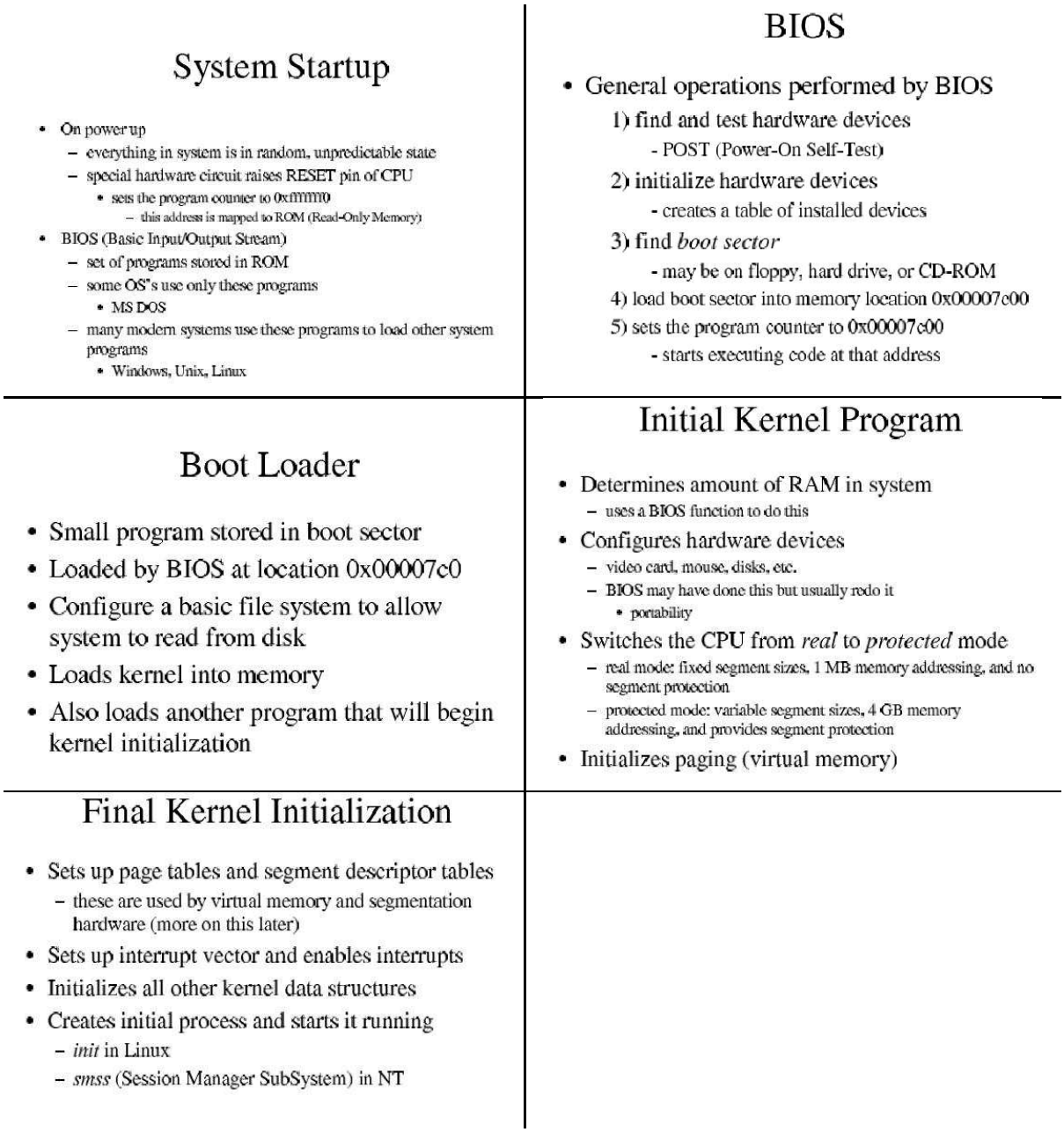
- The initial loading is done by a small program that usually resides in non-volatile memory (e.g., EPROM). This in turn loads the OS from an external device.

- Once loaded, how does the operating system know what to do next? It waits for some event to occur: e.g., the user typing a *command on the keyboard*.

- During "normal" operations of a computer system, some portions of the operating system remain in main memory to provide services for critical operations, such as dispatching, interrupt handling, or managing (critical) resources.

- These portions of the OS are collectively called the *kernel*.

Kernel = OS - transient components
remains *comes and goes*



1.7 System Structure

- An operating system is usually large and complex. Therefore, it should be engineered carefully. Possible ways to structure an operating system:

  – Simple, single-user, *MS-DOS,MacOS, Windows*

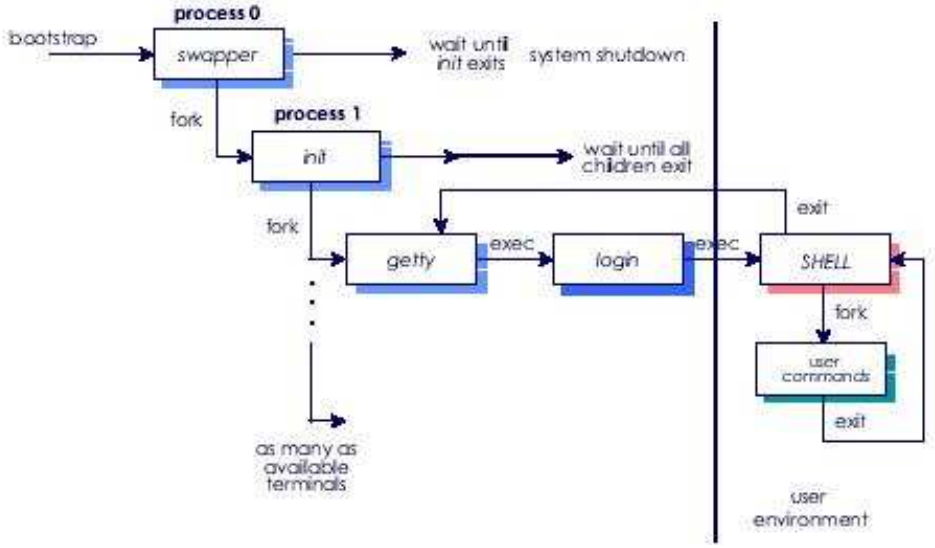


Figure 1.14: UNIN System initilization

- Monolithic, multi-user, *UNIX,Multics, OS/360*
- Layered,*T.H.E. operating system*
- Virtual machine, *IBM VM/370*
- Client/Server (microkernel), *Chorus/MiX*

1.8 Why Study Operating Systems?

- Build or modify real operating system.
- Tune application performance. Understanding the services offered by an operating system will influence how you design applications.
- Administer and use system well. You will develop a better understanding of the structure of modern computing systems, from the hardware level through the operating system level and onto the applications level.
- Can apply techniques used in an OS to other areas;
  - *interesting, complex data structures*
  - *conflict resolution*

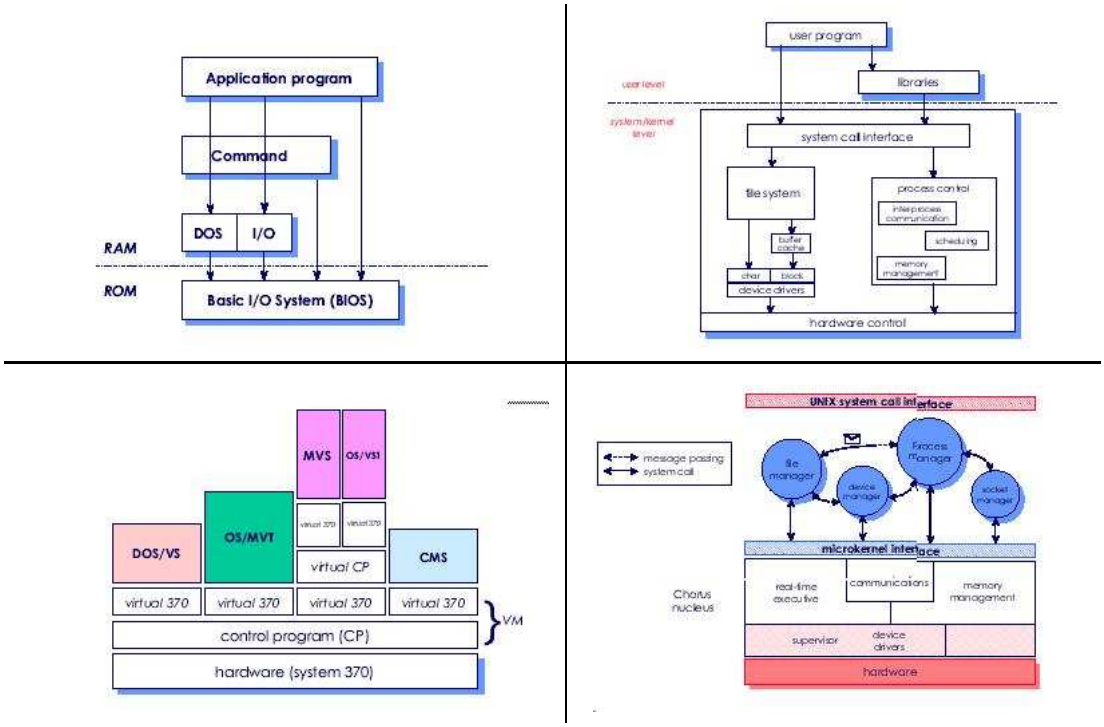


Figure 1.15: OS Structures, MS-DOS, Unix, IBM VM/370, Chorus

  - *concurrency*
  - *resource management*

- Challenge of designing large and complex systems
- Future decisions regarding operating systems will be based on more secure knowledge.
- Curiosity: How the system works.
- *For your Course Requirement!!*

1.8.1 Problems in building OS

- *Large Systems:* 100k's to millions of lines of code involving 100 to 1000 man-years of work
- *Complex:* Performance is important while there is conflicting needs of different users, Cannot remove all bugs from such complex and large software

- Behavior is hard to predict; tuning is done by guessing

Chapter 2

Processes and Threads

Simple C example
Include *text* of *header* file in <> for system, user header name in " "
main program called "main", with these argument types

```
#include <stdio.h>
int main(int argc, char *argv[ ])
{
int i;
for (i=0; i < argc; i++)
printf(‘‘command line argument [%d] = %s \n’’,
i, argv[i]);
}
```

2.1 Processes

What is a Process

- talking about programs executing but what it is meant?
- At the very least, we are recognizing that some program code is resident in memory and the CPU is fetching the instructions in this code and executing them
- Of course, a running program contains data to manipulate in addition to the instructions describing the manipulation. Therefore, there must also be some memory holding data.
- We are starting to talk of processes or tasks or even jobs when referring to the program code and data associated with any particular program

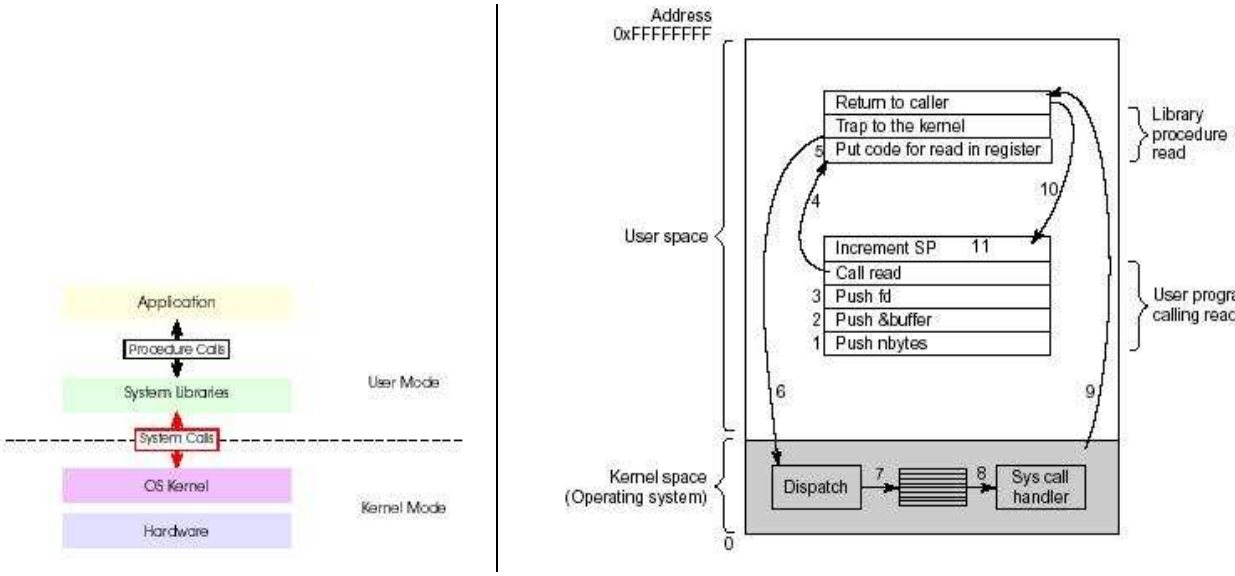


Figure 2.1: System Calls

  - a program in execution,
  - an instance of a program running on a computer,
  - a unit of execution characterised by a single, sequential thread of execution,
  - a current state an associated set of system resources (memory, devices, files),
  - process execution must progress in sequential fashion

- An operating system executes a variety of programs:
  - Batch system, jobs
  - Time-shared systems, user programs or tasks

- Keep track of the states of every process currently executed. make sure; no process monopolises the CPU, no process starves

2.1.1 The Process Model

The operating system must know specific information about processes in order to manage and control them. Such information is usually grouped into two categories:

- process state information

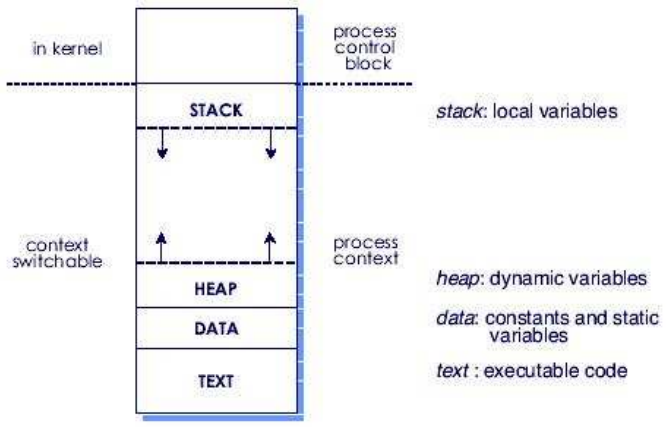


Figure 2.2: A UNIX Process Context

  - CPU registers (general purpose and special purpose); used by the process will include:
    * memory access registers such as a stack pointer and a heap pointer, a stack frame pointer (points at a data block on the stack holding data exchanged between caller and callee functions),
    * a processor status register, possibly a register to hold return addresses,
  - program counter; this is a pointer to the program memory (text) location where the next instruction for this process resides

- process control information
  - scheduling priority, this describes the rules enforced when determining access to a processor by this process, and can include the identity of the "process ready to run" queue that this process is placed in when it is ready to take CPU time
  - resource use information, this information records the use of CPU time, elapsed time, process identity number, user or account identity number, etc.
  - I/O status information, this can include a list of I/O devices used by the process, a list of open files and any buffers associated with them
  - memory allocated, this can describe the region of memory in use (a base address and a size), the page tables (a description of which

pieces of memory are "mapped" into the single region used by the process)

- This collection of process information is represented in the operating system by a data structure element called *process control block (PCB) or a task control block*. Consists of:
  - An executable program (code), which is usually referred to as the text section
  - Associated data needed by the program (global data, stack)
    - the global data variables and constants, which are usually referred to as the data section
    - the dynamic storage memory used to hold temporary variables and pass function call arguments and results, usually referred to as the stack
    - the dynamic storage memory used by C++ new/delete operators and C calls to malloc()/free(), usually referred to as the heap
  - Execution context (or state) of the program;
    - contents of data registers,
    - program counter,
    - stack pointer state (waiting on an event?),
    - memory allocation,
    - status of open files,

2.1.2 Context Switch

- Switching between processes is termed as context switch. When the CPU switches to another process, the system must save the state of the old process and load the saved state for the new process;
  - process table keeps track of processes,
  - context information stored in PCB,
  - process suspended: register contents etc stored in PCB,
  - process resumed: PCB contents loaded into registers
- Context-switch time is overhead; the system does no useful work while switching.

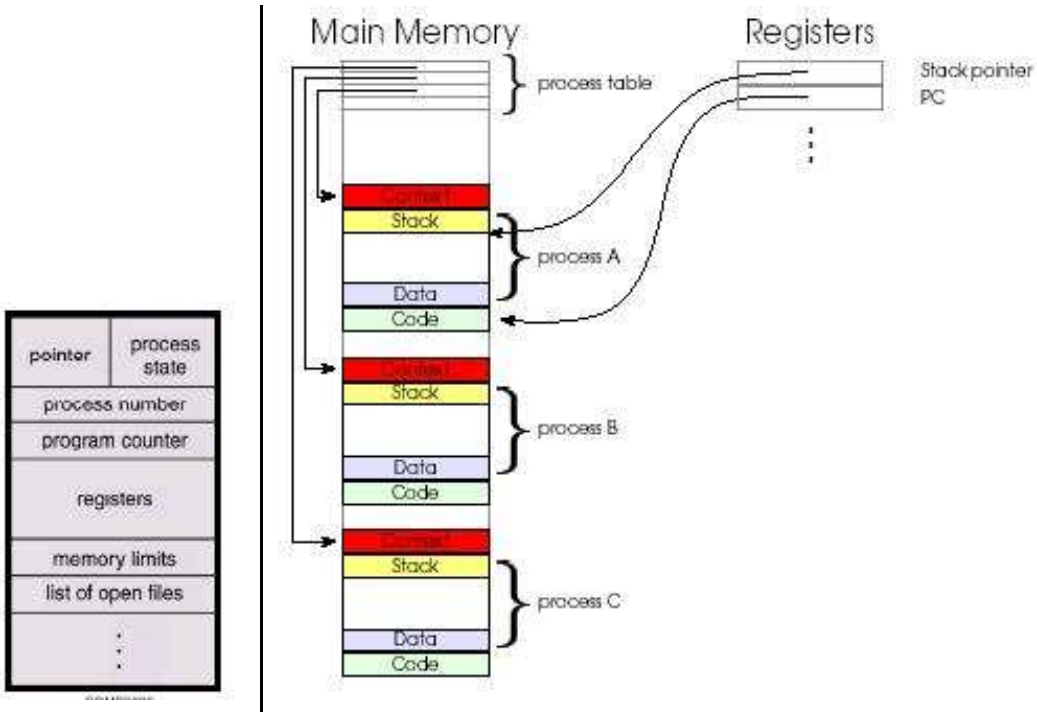


Figure 2.3: Process Control Block (PCB), Processes from main memory to registers.

- Context switching can be critical to performance,
- Dealing with multiple processes is difficult;
  - Synchronization ensure a process waiting for an I/O device receives the signal, signals may be lost or duplicated.
  - Failed mutual exclusion attempt to use a shared resource at the same time.
  - Non-deterministic program operation; program should only depend on input to it, not relying on common memory areas.
  - Deadlocks.
- OS requirements for multiprogramming;
  - Policy to determine which process to schedule (Scheduler).
  - Mechanism to switch between processes (Low-level code that implements the decision Dispatcher).
  - Methods to protect processes from one another (memory system).

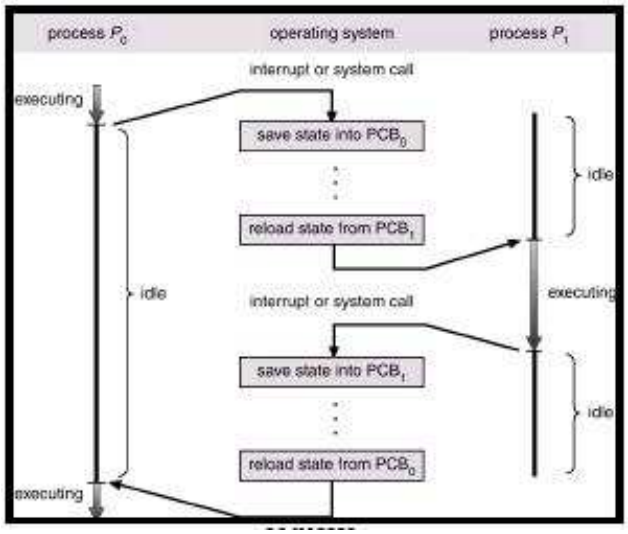


Figure 2.4: CPU Switch From Process to Process

2.1.3 Dispatcher

- Dispatch Mechanism OS keeps system-wide list of processes. Each process in one of three modes; Running: On the CPU (only one in uniprocessor system), Ready: Waiting for the CPU, Blocked: Waiting for I/O or synchronization with another thread. Dispatch loop:

```
while (1) {
run a process for a while
stop process and save its state     context-switch
load state of another process       context-switch
}
```

- How does Dispatcher gain Control?
  - must change from user mode to system mode; the CPU can only do one thing at a time. While a user process is running, dispatcher cannot run, thus the operating system may lose control
  - two ways operating system gains control;
    - Traps: Events internal to user process (System calls, Errors, Page faults)
    - Hardware interrupts: Events external to user process (Character typed at terminal, Completion of disk transfer, Control given to OS interrupt service routine (ISR))

- Dispatcher must track state of process when not running; On every trap or interrupt, save process state in Process Control Block (PCB)

2.1.4 Process Creation

- There are two ways of creating a new process:
  - Build one from scratch:
    - Load *code* and *data* into memory.
    - Create (empty) a *dynamic memory workspace (heap).*
    - Create and initialize the *PCB* (make look like context-switch).
    - Make process known to dispatcher.
  - Clone an existing one (e.g., Unix fork() syscall):
    - Stop current process and save its state.
    - Make a copy of *code*, *data*, *dynamic memory workspace* and *PCB*.
    - Make process known to dispatcher.
- Who creates the processes and how they are supported? *Every operating system has a mechanism to create processes.*
- in UNIX, the **fork()** system call is used to create processes. **fork()** creates an identical copy of the calling process. After the **fork()**, the *parent* continues running concurrently with its *child* competing equally for the CPU. **exec** system call used after a **fork** to replace the process' memory space with a new program.

```
cmd = readcmd();
pid = fork();
if (pid == 0) {
// Child process -- Setup environment here
// e.g., standard i/o, signals exec(cmd);
// exec doesn t return
} else {
// Parent process -- Wait for child to finish
wait(pid);
}
```

- in MS-DOS, the **LOAD_AND_EXEC** system call creates a child process. This call suspends the parent until the child has finished execution, so the parent and child do not run concurrently

- Parent process create children processes, which, in turn create other processes forming a tree of processes
- Resource sharing
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Execution, once a parent creates a child process, a number of execution possibilities exist:
  - Parent and children execute concurrently.
  - the parent may immediately enter a wait state for the child to finish – on UNIX, see the man pages for {wait, waitpid, wait4, wait3}.
  - the parent could immediately terminate.
- If the parent happens to terminate before the child has returned its value, then the child will become a zombie process and may be listed as such in the process status list!
- Address space, once a parent creates a child process, a number of memory possibilities exist:
  - the child can have a duplicate of the parent's address space – as each process continues to execute, their data spaces will presumably diverge.
  - the child can have a completely new program loaded into its address space.
- If either process needs to run a different program, it can perform a call to int execlp(const char *file, const char *arg, ...) where arguments specify the executable file and optional run-time arguments which the caller may wish to provide. See the man pages on fork and also {execl, execlp, execle, exect, execv, execvp}.
- How does each process know whether it is the parent or child after a fork? On BSD UNIX, fork() returns a value of 0 to the child process and returns the process ID of the child process to the parent process.

2.1.5 Process Termination

- A process enters the *exiting* state for one of the following reasons:
  - normal completion: Once a process executes its final instruction, a call to **exit()** is made.
  - abnormal termination: programming errors.
  - run time.
  - I/O.
  - user intervention.
- Even if the user did not program in a call to exit(), the compiler will have appended one to int main()
  - The final result of the process from its int main() is returned to the parent, with a call to wait() if necessary.
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.

2.1.6 Process States

- There are a number of *states* that can be attributed to a process: indeed, the operation of a multiprogramming system can be described by a state transition diagram on the process states. The states of a process include:
  - **New**–a process being created but not yet included in the pool of executable processes (*resource acquisition*).
  - **Ready**–processes that are prepared to execute when given the opportunity.
  - **Active, Running**–the process that is currently being executed by the CPU.
  - **Blocked, Waiting**–a process that cannot execute until some event occurs, such as completion of an I/O service or reception of a signal.
  - **Stopped**–a special case of **blocked** where the process is suspended by the operator or the user.
  - **Exiting, Terminated**–a process that is about to be removed from the pool of executable processes (*resource release*), a process has finished execution and is no longer a candidate for assignment to a processor, and its remaining resources and attributes are to be disassembled and returned to the operating system's "free" resource structures.
- As a process executes, it can change state due to either an external influence, e.g. it is forced to give up the CPU so that another process can take a turn, or an internal reason, e.g. it has finished or is waiting for a service from the operating system
- A process therefore takes part in a finite state system, and we typically show this in a state diagram which highlights the conditions necessary to transit from one state to another

2.2 Threads

- Process: Owner of resources allocated for individual program execution, can encompass more than one thread of execution

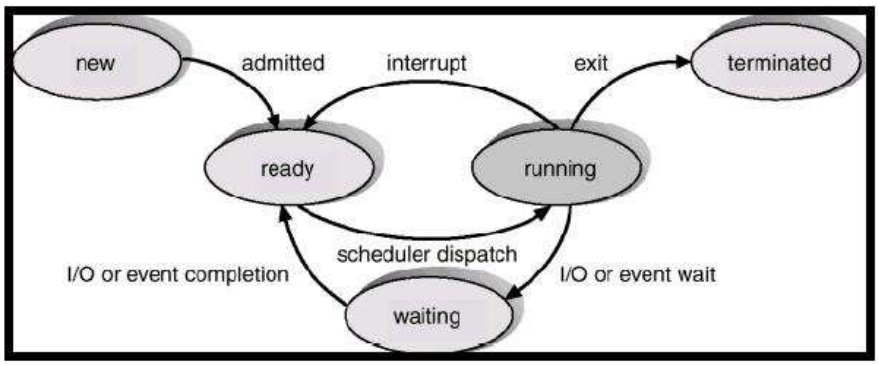


Figure 2.5: Diagram of Process State

- Thread: Unit of execution (unit of dispatching) and a collection of resources, with which the unit of execution is associated, characterize the notion of a process. A *thread* is the abstraction of a unit of execution. It is also referred to as a *light-weight process LWP* that share the same text (program code) and global data, but possess their own CPU register values and their own dynamic (or stack based) variables
- First look at the advantages of threads;
  - a program does not stall when one of its operations blocks.
  - save contents of a page to disk while downloading other page (for web server example)
  - Simplification of programming model
- Single process, single thread MS-DOS, old MacOS
- Single process, multiple threads OS/161
- Multiple processes, single thread traditional Unix
- Multiple processes, multiple threads modern Unices (Solaris, Linux), Windows2000
- As a basic unit of CPU utilization, a thread consists of an instruction pointer (also referred to as the PC or instruction counter), CPU register set and a stack. A thread shares its code and data, as well as system resources and other OS related information, with its peer group (other threads of the same process)

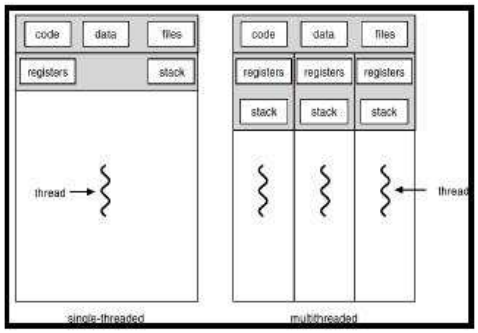


Figure 2.6: Single and Multithreaded Processes

- Threads versus processes;
  - A thread operates in much the same way as a process:
    - can be one of the several states.
    - executes sequentially (within a process and shares the CPU).
    - can issue system calls.
  - Economy of Overheads – managing processes is considerably more expensive than managing threads so LWPs are better.
  - Responsiveness – less setup work means faster response to requests, and multiple thread of execution mean there can be response from some threads even if other threads are busy or blocked.
  - Resource Sharing – Threads within a process share resources (including the same memory address space) conveniently and efficiently compared to separate processes
  - Threads within a process are NOT independent and are NOT protected against each other
  - Multiprocessor Use – if multiple processors are available, a multithreaded application can have its threads run in parallel which means better utilization (especially if there are few other processes present so that, without a multithreaded application, some CPUs would be idle)

- A process utilizing multithreading is a process with multiple points of execution–up to now, we have assumed that each process has only one point of execution

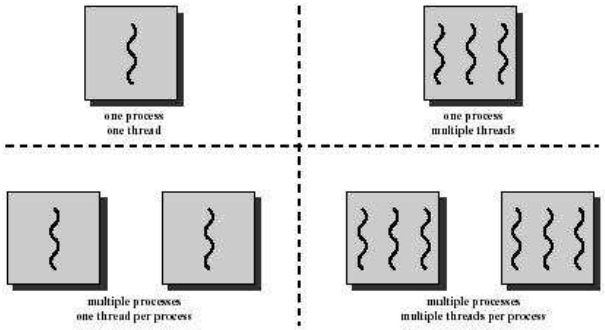


Figure 2.7: Threads and Processes

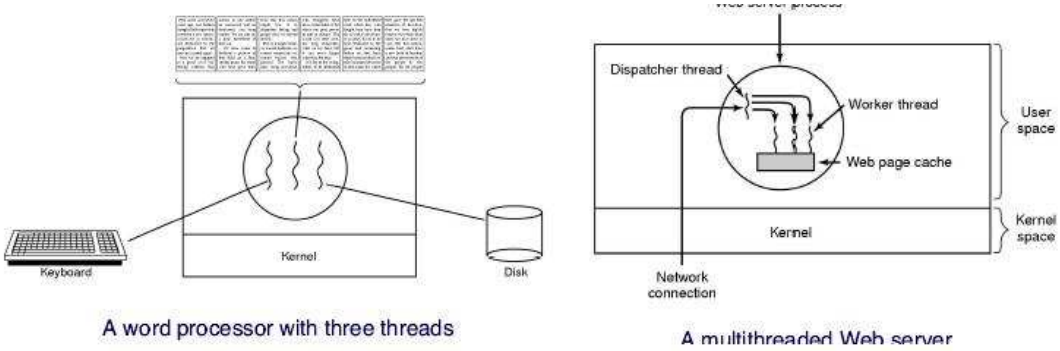


Figure 2.8: A word processor with three threads, a multithreaded web server

- similarity between an operating system supporting multiple processes and a process supporting multiple threads
- Figure 2.6 shows a traditional (or heavyweight) process, on the left, and 3 LWPs are drawn on the right in a way to emphasize their common text and global data (i.e. data and heap)
- In practice, an application such as a web server that can have considerable variations in the rate of requests can create additional threads in response to serving load, yet minimize process creation load on the host
- Less setup improves responsiveness, and shared text means more efficient memory use

2.2.1 The Thread Model

- Many-to-One
  - Many user-level threads mapped to single kernel thread.
  - Thread management is done in user space but the whole process blocks if any one user thread blocks.
  - Used on systems that do not support kernel threads.
- One-to-One
  - Each user-level thread maps to kernel thread.
  - As thread management is done in kernel space, a blocked thread does not prevent other threads from running and multiprocessor utilization is efficient.
  - Examples, Windows 95/98/NT/2000, OS/2
- Many-to-Many
  - Allows many user level threads to be mapped to many kernel threads.
  - The number of kernel threads provided might be specified according to the application and also the number of processors on a particular host.
  - Allows the operating system to create a sufficient number of kernel threads.
  - Solaris 2, DEC/compaq (Thu64), HP (HP-UX), and Silicon Graphics (IRIX), Windows NT/2000 with the *ThreadFiber* package

2.2.2 Implementing Threads in User Space

- Thread management done by user-level threads library
- a thread library is used for management with no support from (or knowledge by) the kernel. If the kernel is single threaded, and one of the user threads blocks, then the user's process is also blocked which means that the remaining user threads are also blocked. Available for many OSes

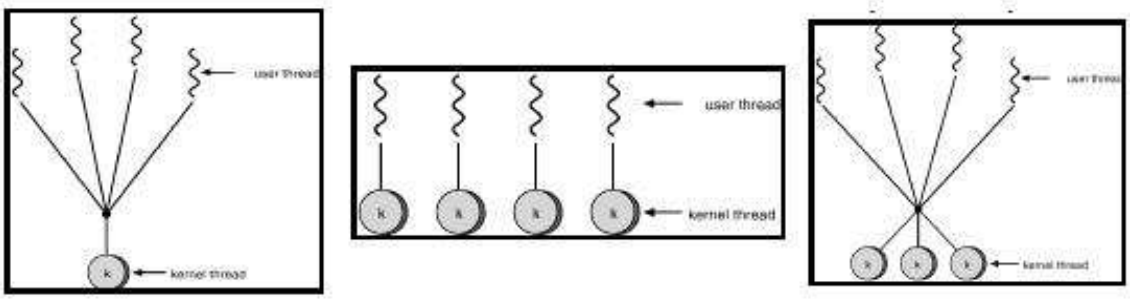


Figure 2.9: Thread Models; Many-to-One, One-to-One, Many-to-Many

- While user threads usually emphasize their lower management load compared to kernel threads, one must consider this in relation to their lower functionality
- Examples;
  - One quite common library is *pthread* – the POSIX (POSIX is an IEEE standard for a portable operating system interface based on UNIX) thread functions.
  - Mach *C-threads*.
  - The Sun Microsystems Solaris 2 OS provides UI-*threads* (a standard originating from the Unix International Organization, RIP).

2.2.3 Implementing Threads in the Kernel

- Supported by the kernel, the kernel performs all management (creation, scheduling, deletion, etc.)
- if one thread blocks, another may be run
- If the kernel is managing multiple processors, an efficient mapping of threads to processors is possible
- Examples; Windows 95/98/NT/2000, Solaris, Tru64 UNIX, BeOS, Linux

2.3 Interprocess Communication

- Multi-threading: concurrent threads share an address space

- Multi-programming: concurrent processes execute on a uniprocessor
- Multi-processing: concurrent processes on a multiprocessor
- Distributed processing: concurrent processes executing on multiple nodes connected by a network
- Concurrent processes (threads) need special support:
  - Communication among processes
  - Allocation of processor time
  - Sharing of resources
  - Synchronization of multiple processes
- In a multiprogramming environment, processes executing concurrently are either *competing* for the CPU and other global resources, or *cooperating* with each other for sharing some resources
- An OS deals with competing processes by carefully allocating resources and properly isolating processes from each other. For cooperating processes, on the other hand, the OS provides mechanisms to share some resources in certain ways as well as allowing processes to properly interact with each other
- Cooperation is either by implicit sharing or by explicit communication
- Processes: *competing* Processes that do not exchange information cannot affect the execution of each other, but they can compete for devices and other resources. Such processes do not intend to work together, and so are unaware of one another
- Properties: Deterministic, Reproducible,Can stop and restart without "side" effects, Can proceed at arbitrary rate
- Processes: *cooperating* Processes that are aware of each other, and directly (by exchanging messages) or indirectly (by sharing a common object) work together, may affect the execution of each other
- Properties: Share (or exchange) something: a common object (or a message), Non-deterministic (a problem!), May be irreproducible (a problem!), Subject to race conditions (a problem!)
- Threads of a process usually do not compete, but cooperate

Table 2.1: Race Condition

Process A	**Process B**	**concurrent access**
A = 1;	**B = 2;**	*does not matter*
A = B + 1;	**B = B * 2;**	*important!*

- Why cooperation? We allow processes to cooperate with each other, because we want to:
  - share some resources.
  - do things faster
    - Read next block while processing current one.
    - Divide jobs into smaller pieces and execute them concurrently.
  - construct systems in modular fashion.
  - UNIX example:

```
cat infile | tr ' ' '\012' |tr '[A-Z]' '[a-z]' | sort | uniq -c
```

2.3.1 Race Conditions

- A potential problem; Instructions of cooperating processes can be interleaved arbitrarily. Hence, the order of (some) instructions are irrelevant. However, certain instruction combinations must be eliminated. For example: see Table 2.1
- A *race condition* is a situation where two or more processes access shared data concurrently and correctness depends on specific interleavings of operations; final value of shared data depends on *timing* (i.e., *race* to access and modify data)
- To prevent race conditions, concurrent processes must be **synchronized**

2.3.2 Critical Regions

- A *section of code*, or a *collection of operations*, in which only one process may be executing at a given time and which we want to make "sort of" atomic. *Atomic* means either an operation happens in its entirely (everything happens at once) or NOT at all; i.e., it cannot be

interrupted in the middle. Atomic operations are used to ensure that cooperating processes execute correctly. Mutual exclusion mechanisms are used to solve the *critical region* problem

- machine instructions are atomic, high level instructions are not (count++; this is actually 3 machine level instructions, an interrupt can occur in the middle of instructions)
- Fundamental requirements; Concurrent processes should meet the following requirements in order to cooperate correctly and efficiently using shared data:
  - *Mutual exclusion*–no two processes will simultaneously be inside the same critical region (CR).
  - *No assumptions*–may be made about speeds or the number of CPUs. Must handle all possible interleavings.
  - *Fault tolerance*–processes running outside their CR should not block with others accessing the CR.
  - *Progress*–no process should have to wait forever to enter its CR. A process wishing to enter its CR will eventually do so in finite time.

  Also, a process in one CR should not block others entering a different CR. *Efficiency*–a process will remain inside its CR for a short time only, without blocking.
- Conceptually, there are three ways to satisfy the implementation requirements:
  - Software approach: put responsibility on the processes themselves
  - Systems approach: provide support within operation system or programming language
  - Hardware approach: special-purpose machine instructions

2.3.3 Mutual Exclusion with Busy Waiting (Software approach)

- *Mutual exclusion* is a mechanism to ensure that only one process (or person) is doing certain things at one time, thus avoid data inconsistency. All others should be prevented from modifying shared data until the current process finishes

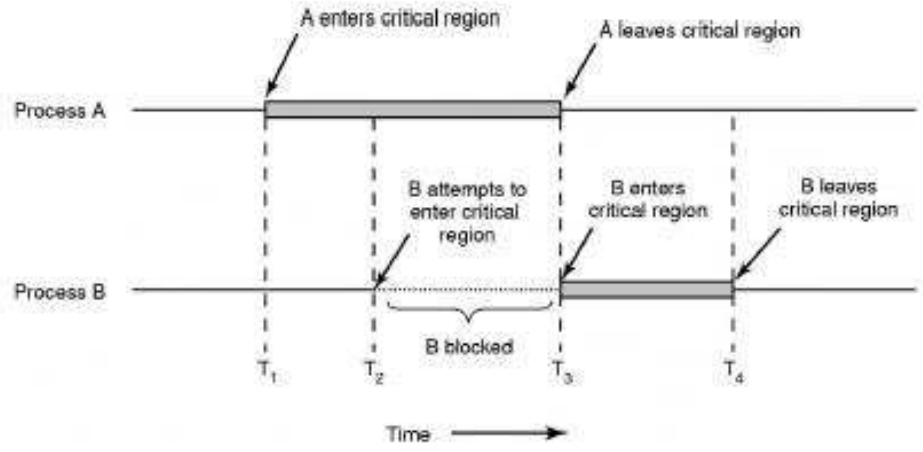


Figure 2.10: Mutual exclusion using critical regions

- Strict Alternation (see Fig. 2.11)
  - the two processes strictly alternate in entering their CR
  - the integer variable **turn,** initially 0, keeps track of whose turn is to enter the critical region
  - **busy waiting**, continuously testing a variable until some value appears, a lock that uses busy waiting is called a **spin lock**
  - both processes are executing in their noncritical regions
  - process 0 finishes its noncritical region and goes back to the top of its loop
  - unfortunately, it is not permitted to enter its CR, **turn** is 1 and process 1 is busy with its nonCR
  - this algorithm does avoid all races
  - but violates condition 3

- Petersons's solution (see Fig. 2.12)
  - does not require strict alternation
  - this algorithm consists of two procedures
  - before entering its CR, each process calls **enter_region** with its own process number, 0 or 1

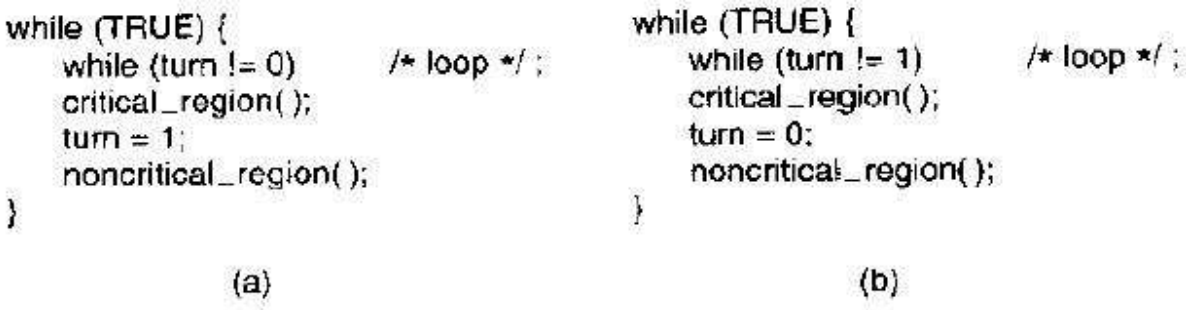


Figure 2.11: A proposed solution to the CR problem. (a) Process 0, (b) Process 1

- after it has finished with the shared variables, the process calls **leave_region** to allow the other process to enter
- consider the case that both processes call **enter_region** almost simultaneously
- both will store their process number in **turn .** Whichever store is done last is the one that counts; the first one is overwritten and lost
- suppose that process 1 stores last , so **turn** is 1.
- when both processes come to the **while** statement, process 0 enters its critical region
- process 1 loops until process 0 exists its CR
- no violation, implements mutual exclusion
- burns CPU cycles (requires busy waiting), can be extended to work for n processes, but overhead, cannot be extended to work for an unknown number of processes, unexpected effects (i.e.,**priority inversion problem**)

2.3.4 Sleep and wakeup

- blocks instead of wasting CPU time (while loop) when they are not allowed to enter their CRs
- *sleep* and *wakeup* pair
- *sleep* is a system call that causes the caller to block (be suspended until another process wakes is up)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 - process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Figure 2.12: Peterson' solution for achieving mutual exclusion

- The Producer-Consumer Problem
  - Suppose one process is creating information that is going to be used by another process, e.g., suppose one process reads information from the disk, and another compiles that information from source to machine code.
  - Producer: creates copies of a resource
  - Consumer: uses up copies of a resource
  - Buffers: used to hold information after producer has created it but before consumer has used it
  - Signaling: keeping control of producer and consumer (e.g., preventing overrun of the producer)
  - Constraints:
    * Consumer must wait for a producer to fill buffers. (signaling)
    * Producer must wait for consumer to empty buffers, when all buffer space is in use. (signaling)
    * Only one process must manipulate buffer pool at once. (mutual exclusion)

- Trouble arises when the producer wants to put a new item in the buffer, but it is already full
- The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items
- RACE CONDITION can occur because access to *count* (see Fig. 2.13) is unconstrained.
  - the buffer is empty
  - the consumer has read *count* to see if it is 0, **sleeping**
  - at that instant, the scheduler started running the producer
  - the producer inserts an item in the buffer, *count* is 1
  - the consumer should be awaken up, the producer calls **wakeup**
  - the consumer is not logically asleep, so the **wakeup** signal is lost
  - the producer will fill up the buffer and also go to sleep
  - BOTH WILL SLEEP FOREVER.

```
#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 2.13: The producer-consumer problem with a fatal race problem

2.3.5 Semaphores

- Dijkstra (1965) introduced the concept of a semaphore
- A semaphore is an integer variable that is accessed through two standard atomic operations: wait (a spinlock, i.e. stops blocking and decrements thesemaphore) and signal (i.e. the semaphore counts the signals it receives)
- Semaphores are variables that are used to signal the status of shared resources to processes (a semaphore could have the value of 0, indicating that no wakeups are saved, or some positive value if one or more wakeups are pending)
- How does that work?
  - If a resource is not available, the corresponding semaphore blocks any process waiting for the resource
  - Blocked processes are put into a process queue maintained by the semaphore (avoids busy waiting!)
  - When a process releases a resource, it signals this by means of the semaphore
  - Signalling resumes a blocked process if there is any
  - Wait and signal operations cannot be interrupted
  - Complex coordination can be specified by multiple semaphores
- the *down* operation on a semaphore
  - checks to see if the value is greater than 0
  - if so, it decrements the value and continues
  - if the value is 0, the process is put to *sleep* without the completing the *down* for the moment
  - all is done as a single, indivisible atomic action
    - checking the value
    - changing it
    - possibly going to sleep
  - it is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed

  - synchronization and no race condition

- the *up* operation on a semaphore
  - increments the value of the semaphore
  - if one or more processes were sleeping on that semaphore, unable to complete an earlier *down* operation, one of them is chosen by the system and allowed to complete its *down*
  - the semaphore will be 0. but there will be one fewer process sleeping on it
  - indivisible process; incrementing the semaphore and waking up one process

- Solving the producer-consumer problem using semaphores (see Fig. 2.14)
  - the solution uses three semaphores;
    * one called **full** for counting the number of slots that are full
    * one called **empty** for counting the number of slots that are empty
    * one called **mutex** to make sure the producer and the consumer do not access the buffer at the same time. **mutex** is initially 1 (**binary semaphore**)
    * if each process does a *down* just before entering its CR and an *up* just after leaving it, the mutual exclusion is guaranteed.

- Possible uses of semaphores;
  - Mutual exclusion, initialize the semaphore to one
  - Synchronization of cooperating processes (signaling), initialize the semaphore to zero
  - Managing multiple instances of a resource, initialize the semaphore to the number of instances

- Type of semaphores;
  - **binary** is a semaphore with an integer value of 0 and 1.
  - **counting** is a semaphore with an integer value ranging between 0 and an arbitrarily large number. Its initial value might represent the number of units of the critical resources that are available. This form is also known as a general semaphore.

```
#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 2.14: The producer-consumer problem using semaphore

2.3.6 Monitors

- Semaphores are useful and powerful
- But, they require programmer to think of every timing issue; easy to miss something, difficult to debug
- Let the compiler handle the details
- Monitors are a high level language construct for dealing with synchronization
  - similar to classes in Java
  - a monitor has fields and methods
- A monitor is a software module implementing mutual exclusion
- Monitors are easier to program than semaphores
  - programmer only has to say what to protect
  - compiler actually does the protection (compiler will use semaphores to do protection)
- Natively supported by a number of programming languages: Java
  - Resources or critical sections can be protected using the keyword: **synchronized** keyword
  - **synchronized** can be applied to a method: entire method is a critical section
- Chief characteristics (see Fig. 2.15):
  - Local data variables are accessible only by the monitor (not externally)
  - Process enters monitor by invoking one of its procedures, but cannot directly access the monitor's internal data structures
  - Only one process may be executing in the monitor at a time (mutual exclusion)
  - Only methods inside monitor can access fields
  - At most one thread can be active inside monitor at any one time
- Main problem: provides less control

- Allow process to wait within the monitor with **condition** variable, condition x,y;
- can only be used with operations **wait** and **signal** (notify() in Java);
  - operation **wait(x)**; means that the process invoking this operation is suspended until another process invokes **signal(x)**;
  - operation **signal(x)**; resumes exactly one process suspended
- condition variables are not counters, they do not accumulate signals for later use the way the semaphores do. Thus if a condition variable is signaled with no waiting on it, the signal is lost
- This solution is deadlock free
- In Fig. 2.16, the solution for the producer-consumer problem with a monitor is given
- The class **our_monitor** contains the buffer, the administration variables and two synchronized methods
- when the producer is active in *insert*, it knows for sure that the consumer can not be active inside *remove*
- making it safe to update the variables and buffer without fear of race conditions

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

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

Figure 2.15: A monitor

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Figure 2.16: The producer-consumer problem with a monitor.

2.4 Classical IPC Problems

2.4.1 The Dining Philosophers Problem (see Fig. 2.17)

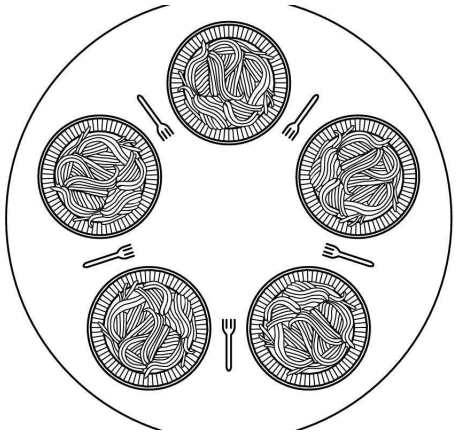


Figure 2.17: Lunch time in the Philosophy Department.

- Five philosophers are seated around a circular table
- A philosopher needs two forks to eat
- The life of a philosopher consists of alternate periods of eating and thinking
- Write a program for each philosopher that does what it is supposed to do and never gets stuck
- one attempt is to use a binary semaphore (**think**)
  - before starting to acquire forks, a philosopher would do a **down** on *mutex*
  - after replacing the forks, he would **up** on *mutex*
  - bug: only one philosopher can be eating at any instant
- the solution presented in Fig. 2.18 uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks)
- A philosopher may move only into eating state if neither neighbor is eating

```
#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 2.18: A solution to the dining philosophers problem.

- The solution is deadlock–free and allows the maximum parallelism for any number of philosophers

2.4.2 The Readers and Writers Problem (see Fig. 2.19)

```
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 (rc == 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 2.19: A solution to the readers and writers problem.

- Models access to a database; many competing processes wishing to read and write

- 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 readers

- Write a program for the readres and writers

- the solution presented in Fig. 2.19, the first reader to get access to the database does a **down** on the semaphore *db*
- Subsequent readers increment a counter, *rc*
- As readers leave, they decrement the counter and the last one does an **up** on the semaphore, allowing a blocked writer
- bug:
  - As long as at least one reader is still active, subsequent readers is 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
- The solution is that when a reader arrives and a write is waiting, the reader is suspended behind the writer instead of being admitted immediately (less concurrency, lower performance)

2.4.3 The Sleeping Barber Problem (see Fig. 2.20)

- This problem is similar to various queueing situations
- The problem is to program the barber and the customers without getting into race conditions
  - Solution uses three semaphores:
    - *customers*; counts the waiting customers
    - *barbers*; the number of barbers (0 or 1)
    - *mutex*; used for mutual exclusion
    - also need a variable *waiting*; also counts the waiting customers (reason; no way to read the current value of semaphore)
  - The barber executes the procedure *barber*, causing him to block on the semaphore *customers* (initially 0)
  - The barber then goes to sleep
  - 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*

```
#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)
{
     while (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 2.20: A solution to the sleeping barber problem.

– 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*

– When the customer releases *mutex*, the barber begins the haircut

2.5 Scheduling

- In multiprogramming systems, where there is more than one process runnable (i.e., ready), the operating system must decide which one to run next
- The decision is made by the part of the operating system called the *scheduler*, using a *scheduling algorithm* or *scheduling discipline.*

2.6 Introduction to Scheduling

- **In the beginning**–there was no need for scheduling, since the users of computers lined up in front of the computer room or gave their job to an operator
- **Batch processing**–the jobs were executed in first come first served manner
- **Multiprogramming**–life became complicated!
- The scheduler is concerned with deciding *policy*, not providing a *mechanism*
- The dispatcher is the mechanism
- Dispatcher
  - Low-level mechanism
  - Responsibility: Context-switch
    - Save execution state of old process in PCB
    - Load execution state of new process from PCB to registers
    - Change scheduling state of process (**running, ready,** or **blocked**)
    - Switch from kernel to user mode
    - Jump to instruction in user process
- Scheduler
  - Higher-level policy
  - Responsibility: Deciding which process to run
- Scheduling refers to a set of policies and mechanisms to control the order of work to be performed by a computer system.

- Of all the resources of a computer system that are scheduled before use, the CPU is the far most important.
- But, other criteria may be important too (e.g.,memory)
- Multiprogramming is the (efficient) scheduling of the CPU
- Metrics
  - Execution time: T_s
  - Waiting time: time a thread waits for execution: T_w
  - Turnaround time: time a thread spends in the system (waiting plus execution time): $T_s + T_w = T_r$
  - Normalized turnaround time: T_r/T_s
- Process Behavior
  - The basic idea is to keep the CPU busy as much as possible by executing a (user) process until it must wait for an event and then switch to another process
  - Processes alternate between consuming CPU cycles (*CPU-burst*) and performing I/O (*I/O-burst*)
- Categories of Scheduling Algorithms (See Fig. 2.21)
  - In general, scheduling policies may be *preemptive* or *non- preemptive*
  - In a non-preemptive pure multiprogramming system, the short-term scheduler lets the current process run until it blocks, waiting for an event or a resource, or it terminates. First-Come-First-Served (FCFS), Shortest Job first (SJF). Good for "background" batch jobs.
  - Preemptive policies, on the other hand, force the currently active process to release the CPU on certain events, such as a clock interrupt, some I/O interrupts, or a system call. Round-Robin (RR), Priority Scheduling. Good for "foreground" interactive jobs
- Scheduling Algorithm Goals
  - A typical scheduler is designed to select one or more primary performance criteria and rank them in order of importance

All systems
Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems
Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems
Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Figure 2.21: Some goals of the scheduling algorithm under different circumstances.

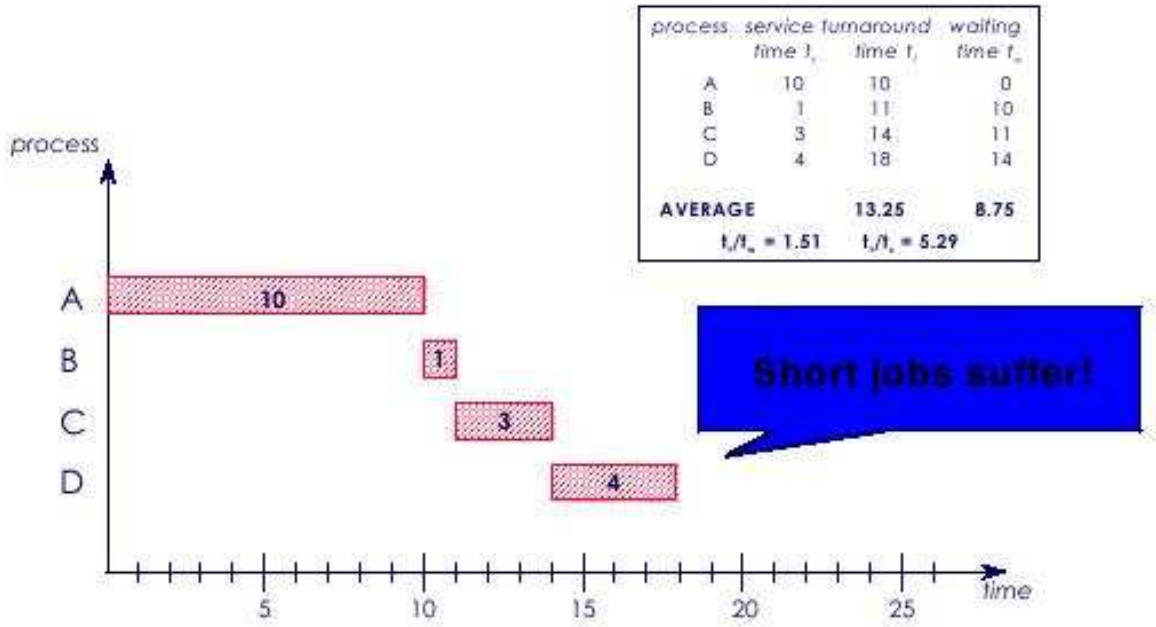
- One problem in selecting a set of performance criteria is that they often conflict with each other
- For example, increased processor utilization is usually achieved by increasing the number of active processes, but then response time decreases
- So, the design of a scheduler usually involves a careful balance of all requirements and constraints
- The following is only a small subset of possible characteristics:*I/O throughput, CPU utilization, response time (batch or interactive), urgency of fast response, priority, maximum time allowed, total time required.*
- Maximize:
  - CPU utilization
  - throughput (number of tasks completed per time unit, also called bandwidth)
- Minimize:
  - Turnaround time (submission to completion, also called latency)
  - Waiting time (sum of time spent in Ready-queue)

    - Response time (time from start of request to production of first response, not full time for output)
  - Fairness:
    - every task should be handled eventually (no starvation)
    - tasks with similar characteristics should be treated equally
  - different type of systems have different priorities!

2.7 Scheduling in Batch Systems

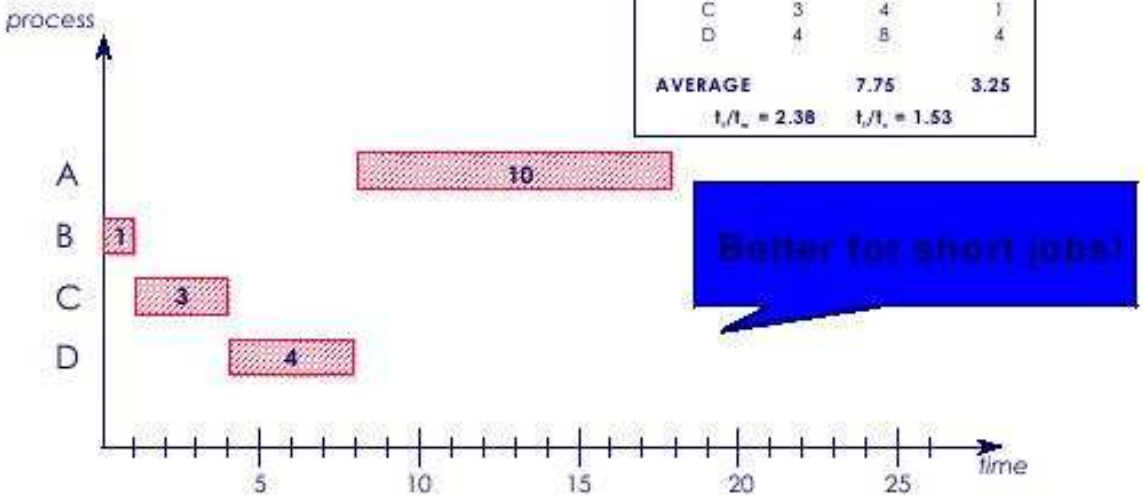
- First-Come First Served (FCFS) (See Fig. 2.22)
  - FCFS, also known as First-In-First-Out (FIFO), is the simplest scheduling policy
  - Arriving jobs are inserted into the tail of the ready queue and the process to be executed next is removed from the head (front) of the queue
  - FCFS performs better for long jobs
  - Relative importance of jobs measured only by arrival time (poor choice)
  - A long CPU-bound job may take the CPU and may force shorter (or I/O-bound) jobs to wait prolonged periods
  - This in turn may lead to a lengthy queue of ready jobs, and thence to the "convoy effect

- Shortest Job First (SJF)(See Fig. 2.23)
  - SJF policy selects the job with the shortest (expected) processing time first
  - Shorter jobs are always executed before long jobs
  - One major difficulty with SJF is the need to know or estimate the processing time of each job (can only predict the future!)
  - Also, long running jobs may starve for the CPU when there is a steady supply of short jobs
  - SJF is optimal Â minimum average waiting time for given set of processes
  - nonpreemptive Â once CPU given to process, can't be preempted until completes CPU burst



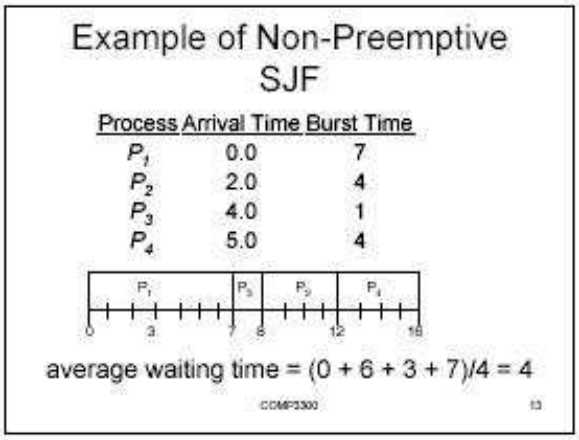
process	service time t_s	turnaround time t_t	waiting time t_w
A	10	10	0
B	1	11	10
C	3	14	11
D	4	18	14
AVERAGE		13.25	8.75
	t_t/t_s = 1.51	t_t/t_s = 5.29	

Figure 2.22: An example to First-Come First Served.



process	service time t_s	turnaround time t_t	waiting time t_w
A	10	18	8
B	1	1	0
C	3	4	1
D	4	8	4
AVERAGE		7.75	3.25
	t_t/t_s = 2.38	t_t/t_s = 1.53	

Figure 2.23: An example to Shortest Job First.



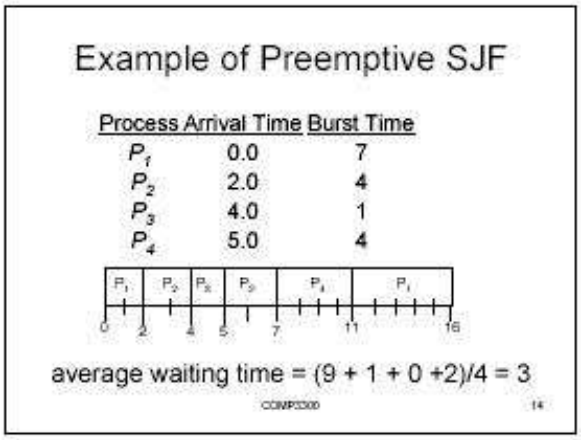


Figure 2.24: Example of non-preemptive SJF and example of preemptive SJF.

- Shortest Remaining Time Next (SRTF)
  - preemptive version of the SJF
  - if new process arrives with CPU burst length remaining time of current executing process, preempt: Shortest-Remaining-Time-First

2.8 Scheduling in Interactive Systems

- Round-Robin Scheduling (RR) (See Fig. 2.25)
  - RR reduces the penalty that short jobs suffer with FCFS by preempting running jobs periodically
  - Scheduled thread is given a time slice
  - The CPU suspends the current job when the reserved *quantum* (*time-slice*) is exhausted
  - The job is then put at the end of the ready queue if not yet completed
  - Advantages;
    - no starvation

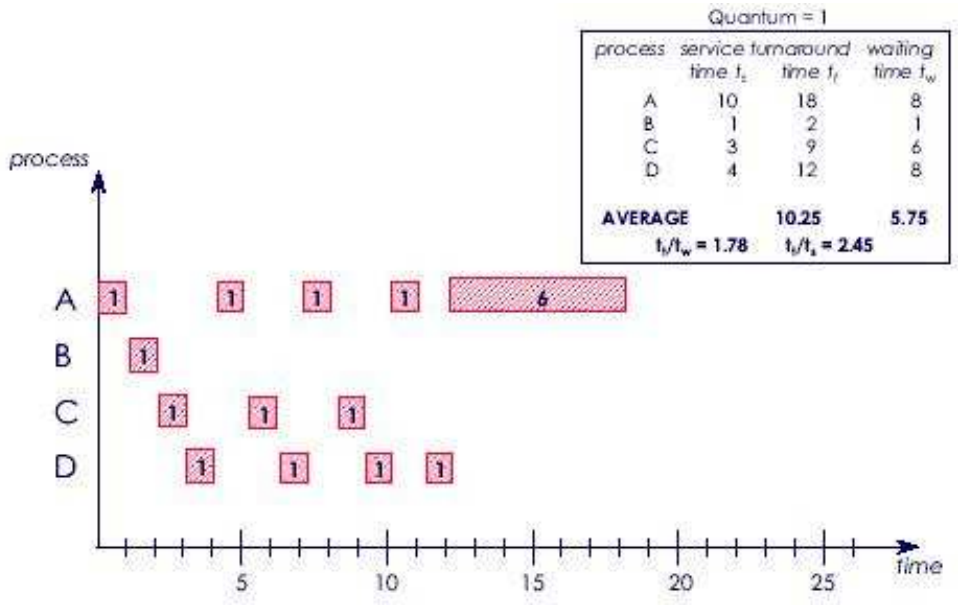


Figure 2.25: An example to Round Robin.

  * Fair allocation of CPU across jobs
  * Low average waiting time when job lengths vary widely
- Disadvantages;
  * Poor average waiting time when job lengths are identical; Imagine 10 jobs each requiring 10 time slices, all complete after about 100 time slices, even FCFS is better!
  * The critical issue with the RR policy is the length of the quantum. If it is too short, then the CPU will be spending more time on context switching. Otherwise, interactive processes will suffer

- Priority Scheduling (See Fig. 2.26)
  - Each process is assigned a priority (e.g., a number)
  - The ready list contains an entry for each process ordered by its priority
  - The process at the beginning of the list (highest priority) is picked first
    * Scheduler will always choose a thread of higher priority over one of lower priority

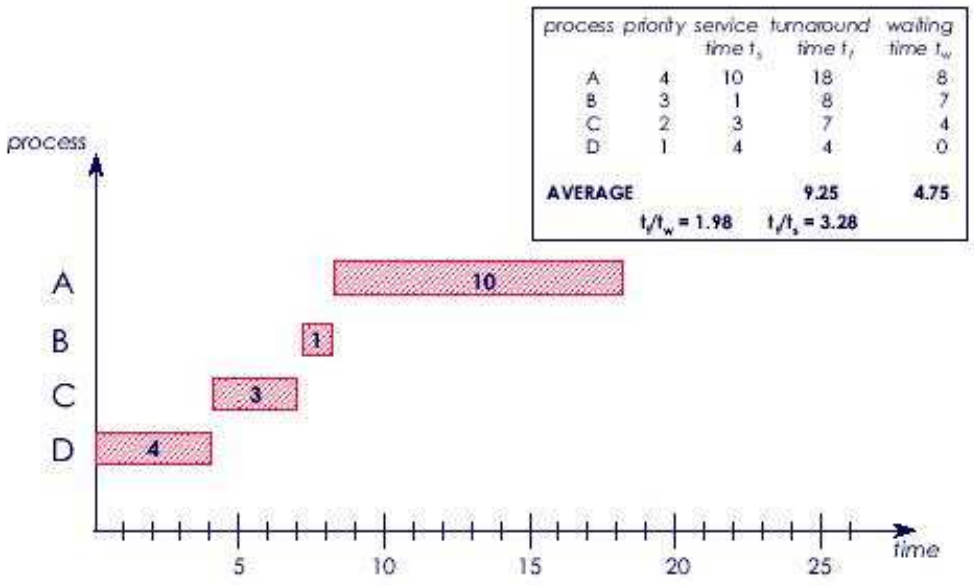


Figure 2.26: An example to Priority-based Scheduling.

  - ∗ Implemented via multiple FCFS ready queues (one per priority)
- – Lower-priority may suffer starvation
- – A variation of this scheme allows preemption of the current process when a higher priority process arrives
- – Another variation of the policy adds an aging scheme where the priority of a process increases as it remains in the ready queue; hence, will eventually execute to completion

• Multiple Queues (See Fig. 2.27)

- – Multi-Level Queue (MLQ) scheme solves the mix job problem (e.g., batch, interactive, and CPU-bound) by maintaining separate "ready" queues for each type of job class and apply different scheduling algorithms to each
- – Multi-level feedback queue
  - ∗ this is a variation of MLQ where processes (jobs) are *not* permanently assigned to a queue when they enter the system

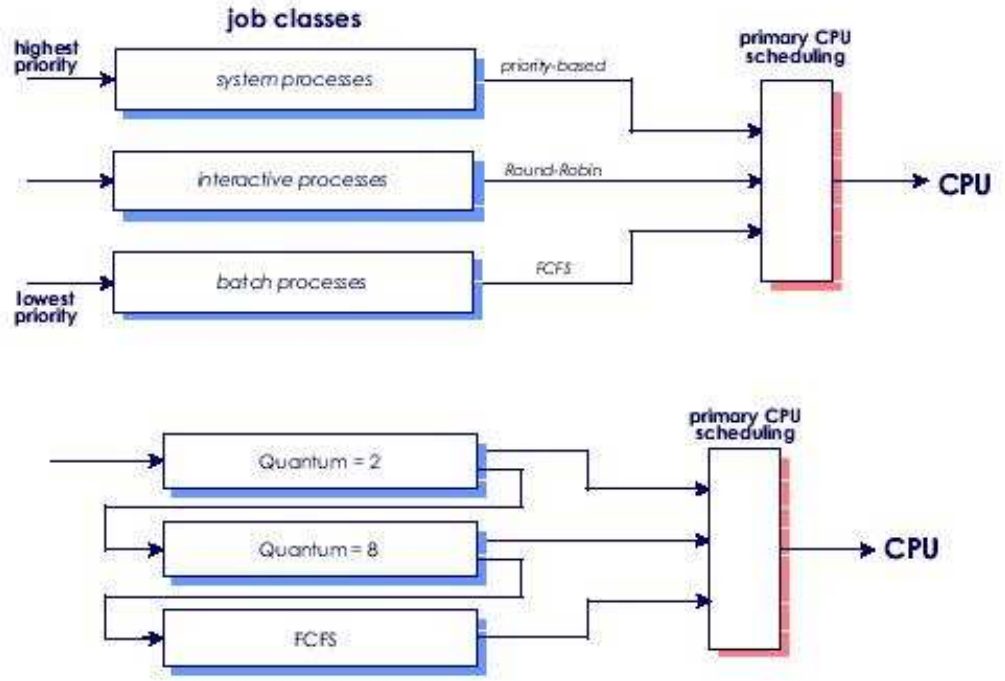


Figure 2.27: Multi-level queue and Multi-level feedback queue (lower).

  - ∗ In this approach, if a process exhausts its time quantum (i.e., it is CPU-bound), it is moved to another queue with a longer time quantum and a lower priority
  - ∗ The last level usually uses FCFS algorithm in this scheme

- Lottery Scheduling
  - – Implemented guaranteed access to resources is, in general, difficult!
  - – process gets "lottery tickets" for various resources
  - – more lottery tickets imply better access to resource
  - – Advantages: Simple, Highly responsive, Allows cooperating processes/threads to implement individual scheduling policy (exchange of tickets)
  - – Process A: 15% of CPU time, Process B: 25% of CPU time, Process C: 5% of CPU time, Process D: 55% of CPU time How many tickets should each process get to achieve this?

2.9 Policy versus Mechanism

- Separate what is allowed to be done with how it is done; a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized; mechanism in the kernel
- Parameters filled in by user processes; policy set by user process

Chapter 3

Deadlock

- **Deadlock** is defined as the *permanent* blocking of a set of processes that compete for system resources, including database records or communication lines
- Unlike other problems in multiprogramming systems, there is no efficient solution to the deadlock problem in the general case
- Deadlock **prevention**, **by design**, is the "best" solution
- Deadlock occurs when a set of processes are in a wait state, because each process is waiting for a resource that is held by some other waiting process
- None will release what they hold until they get what they are waiting for
- Therefore, all deadlocks involve conflicting resource needs by two or more processes
- Example: Unordered Mutex; Two threads accessing two locks

$Semaphore\ m[2] = \{1, 1\};$	$//binarysemaphore$
$Thread1$	$Thread2$
$m[0].P();$	$m[1].P();$
$m[1].P();$	$m[0].P();$
$//access\ shared\ data$	$//access$
$m[1].V();$	$m[0].V();$
$m[0].V();$	$m[1].V();$

- What happens if Thread1 grabs $m[0]$ and Thread2 grabs $m[1]$? (P means down operation and V means up operation)

3.1 Resources

- Classification of resources–I, Two general categories of resources can be distinguished:
  - **Reusable**: something that can be safely used by one process at a time and is not depleted by that use.
    - Processes obtain resources that they later release for reuse by others.
    - Examples are processors, I/O channels, main and secondary memory, files, specific I/O devices, databases, and semaphores.
    - In case of two processes and two resources, deadlock occurs if each process holds one resource and requests the other.
  - **Consumable**: these can be created and destroyed.
    - When a resource is acquired by a process, the resource ceases to exist.
    - Examples are interrupts, signals, messages, and information in I/O buffers
    - Deadlock may occur if a Receive message is blocking
    - May take a rare combination of events to cause deadlock
- Classification of resources–II, One other taxonomy again identifies two types of resources:
  - **Preemptable**: these can be taken away from the process owning it with no ill effects (needs save/restore). E.g., memory or CPU.
  - **Non-preemptable**: cannot be taken away from its current owner without causing the computation to fail. E.g., printer or floppy disk.
- Deadlocks occur when sharing *reusable* and *non-preemptable* resources

3.2 Introduction to Deadlocks

3.2.1 Conditions for Deadlock (See Fig. 3.1)

- Four conditions that must hold for a deadlock to be possible:
  - **1. Mutual exclusion**: processes require exclusive control of its resources (not sharing), only one process may use a resource at a time

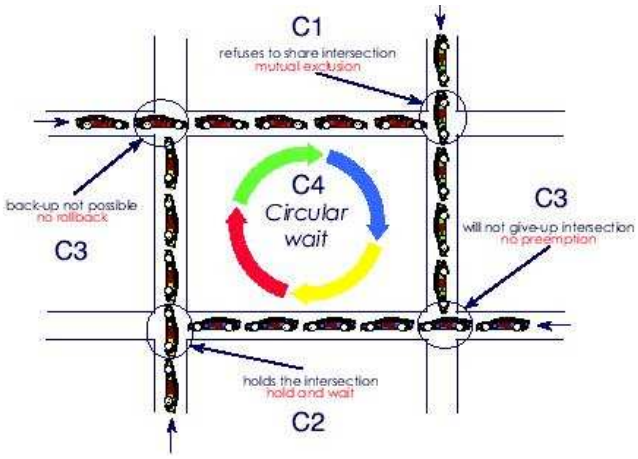


Figure 3.1: An example to Deadlock.

  - **2. Hold and wait**: process may wait for a resource while holding others
  - **3. No preemption**: process will not give up a resource until it is finished with it. Also, **processes are irreversible**: unable to reset to an earlier state where resources not held
  - **4. Circular wait**: each process in the chain holds a resource requested by another, there exists set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 waiting for resource held by P_1, P_1 waiting for resource held by P_2, ..., P_{n-1} waiting for resource held by P_n, P_n waiting for resource held by P_0

- If any one of the necessary conditions is prevented a deadlock need not occur. For example:
  - Systems with only simultaneously shared resources cannot deadlock; Negates *mutual exclusion.*
  - Systems that abort processes which request a resource that is in use; Negates *hold and wait.*
  - Preemptions may be possible if a process does not use its resources until it has acquired all it needs; Negates *no preemption.*
  - Transaction processing systems provide checkpoints so that processes may back out of a transaction; Negates *irreversible process.*
  - Systems that prevent, detect, or avoid cycles; Negates *circular wait.* Often, the preferred solution.

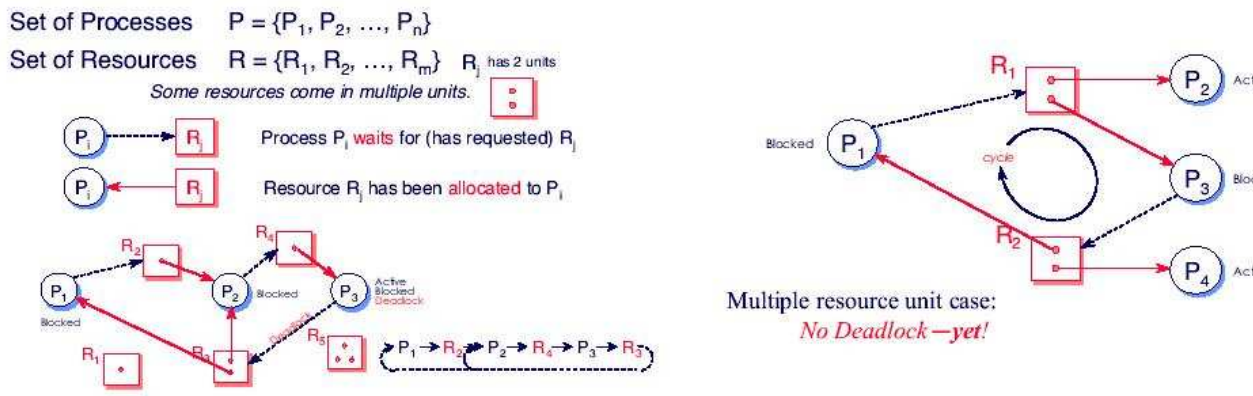


Figure 3.2: Resource Allocation Graphs, in the right one, either P_2 or P_4 could relinquish a resource allowing P_1 or P_3 (which are currently blocked) to continue.

3.2.2 Deadlock Modeling (See Fig. 3.2)

- Cycle is a *necessary condition* for a deadlock
- But when dealing with multiple unit resources –*not sufficient*
- A *knot* must exist–a cycle with no non-cycle outgoing path from any involved node
- At the moment assume that:
  - a process *halts* as soon as it waits for one resource,
  - processes can wait for only *one* resource at a time
- In general, four strategies are used for dealing with deadlocks:
  - **Ignore (The Ostrich Algorithm)**: stick your head in the sand and pretend there is no problem at all.
  - **Prevention**: design a system in such a way that the possibility of deadlock is excluded *a priori* (e.g., compile-time/statically, by design)
  - **Avoidance**: make a decision dynamically checking whether the request will, if granted, potentially lead to a deadlock or not (e.g., run-time/dynamically, before it happens)

  - **Detection and Recovery**: let the deadlock occur and detect when it happens, and take some action to recover after the fact (e.g., run-time/dynamically, after it happens)

3.3 The Ostrich Algorithm

- Different people react to this strategy in different ways:
  - **Mathematicians**: find deadlock totally unacceptable, and say that it must be prevented at all costs.
  - **Engineers**: ask how serious it is, and do not want to pay a penalty in performance and convenience.
- The UNIX approach is just to ignore the problem on the assumption that most users would prefer an occasional deadlock, to a rule restricting user access to only one resource at a time
- The problem is that the prevention price is high, mostly in terms of putting inconvenient restrictions on processes

3.4 Deadlock Detection and Recovery

- This technique does not attempt to prevent deadlocks; instead, it lets them occur
- The system detects when this happens, and then takes some action to recover after the fact (i.e., is reactive)
- With deadlock detection, requested resources are granted to processes whenever possible
- Periodically, the operating system performs an algorithm that allows it to detect the circular wait condition
- A check for deadlock can be made as frequently as resource request, or less frequently, depending on how likely it is for a deadlock to occur
- Checking at each resource request has two advantages: It leads to early detection, and the algorithm is relatively simple because it is based on incremental changes to the state of the system

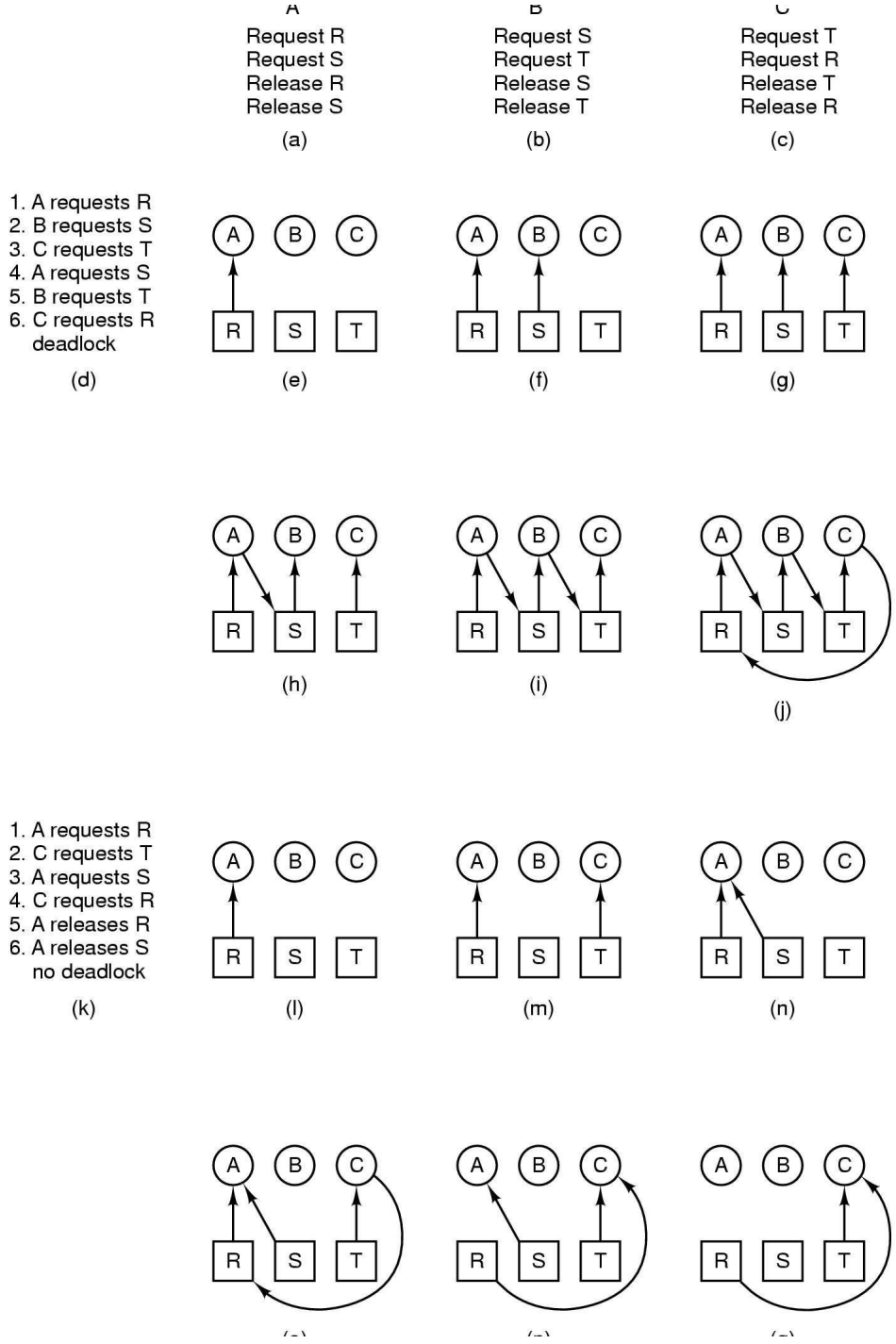


Figure 3.3: An example of how deadlock occurs and how it can be avoided.

- On the other hand, such frequent checks consume considerable processor time
- Once the deadlock algorithm has successfully detected a deadlock, some strategy is needed for recovery
- There are various ways:
  - Recovery through *Preemption*; In some cases, it may be possible to temporarily take a resource away from its current owner and give it to another.
  - Recovery through *Rollback*; If it is known that deadlocks are likely, one can arrange to have processes *checkpointed* periodically. For example, can undo transactions, thus free locks on database records. This often requires extra software functionality.
  - Recovery through *Termination*; The most trivial way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. Warning! *Irrecoverable losses or erroneous results may occur, even if this is the least advanced process.*

3.5 Deadlock Avoidance

- Deadlock avoidance, allows the necessary conditions but makes sensible choices to ensure that a deadlock-free system remains free from deadlock
- With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Deadlock avoidance thus requires knowledge of future requests for process resources
- Ways to avoid deadlock by careful resource allocation:
  - Resource trajectories.
  - Safe/unsafe states.
  - Dijkstra's Banker's algorithm.

3.5.1 Resource Trajectories (See Fig. 3.4)

- The horizontal (vertical) axis represents the number of instructions executed by process **A** (**B**)
- Every point in the diagram represents a joint state of the two processes
- If the system ever enters the box bounded by I_1 and I_2 on the sides and I_5 and I_6 top and bottom, it will eventually deadlock when it gets to the intersection of I_2 and I_6
- At this point, **A** is requesting the plotter and **B** is requesting the printer, and both are already assigned
- The entire box is unsafe and must not be entered
- At point t the only safe thing to do is run process **A** until it gets to I_4. Beyond that, any trajectory to u will do
- At point t **B** is requesting a resource. The system must decide whether to grant it or not
- If the grant is made, the system will enter an unsafe region and eventually deadlock

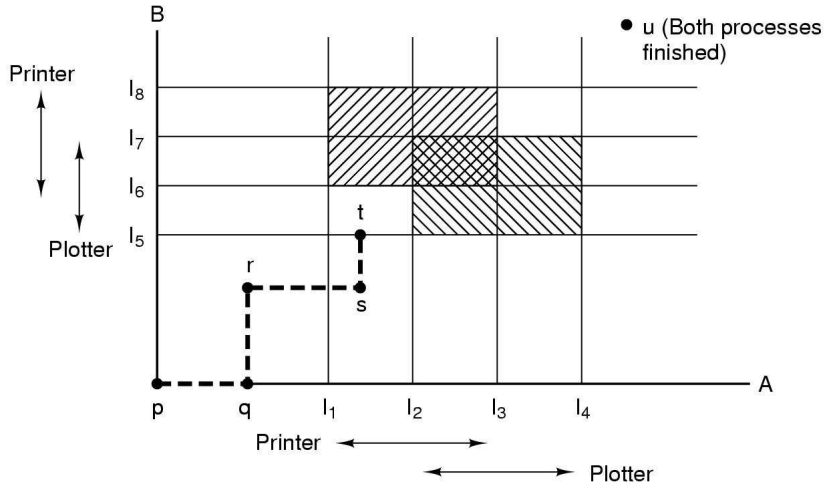


Figure 3.4: Two process resource trajectories.

3.5.2 Safe and Unsafe States (See Fig. 3.5)

- $\sum_{i=1}^{n} C_{ij} + A_j = E_j$
  - **C**: Current Allocation Matrix
  - **A**: Resources Available
  - **E**: Resources in Existence
- Add up all the instances of the resource j that have been allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist
- At any instant of time, there is a current state consisting of **E**, **A**, **C**, and **R** (Request Matrix)
- A state is said to be **safe** if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
- A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free
- The upper state of Fig 3.5 is safe because there exist a sequence of allocations (scheduler runs B) that allows all processes to complete; by careful scheduling, can avoid deadlock

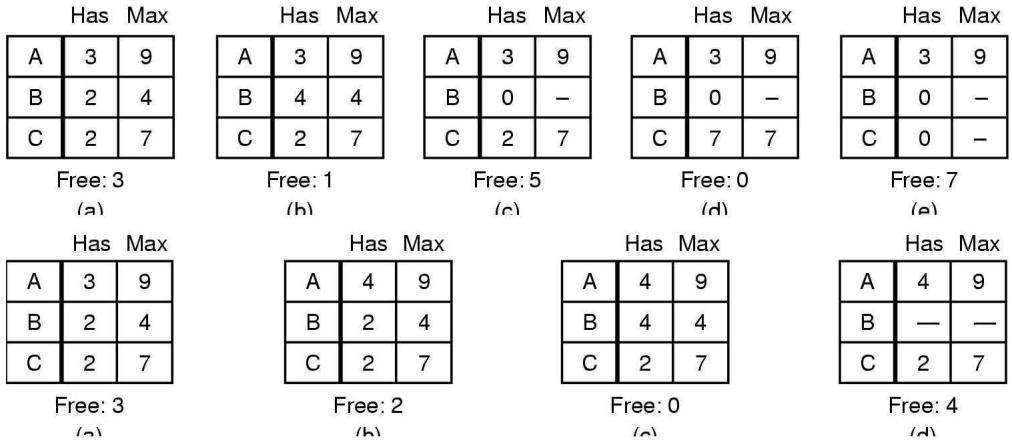


Figure 3.5: Demonstration that the state in is safe (upper), and in is not safe (lower).

- The lower state of Fig 3.5 is not safe because this time scheduler runs **A** and **A** gets another resource
- There is no sequence that guarantees completion
- An unsafe state is not a deadlock state
- The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given

3.5.3 The Banker's Algorithm for Deadlock Avoidance

- Assume **N** Processes P_i, **M** Resources R_j
- Availability vector $Avail_j$, units of each resource (initialized to maximum, changes dynamically)
- Let Max_{ij} be an $N \times M$ matrix
- $Max_{ij} = L$ means Process P_i will request at most L units of R_j
- $Hold_{ij}$ Units of R_j currently held by P_i
- $Need_{ij}$ Remaining need by P_i for units of R_j

- $Need_{ij} = Max_{ij} - Hold_{ij}$, for all i, j
- Resource Request
  - At any instance, P_i posts its request for resources in vector REQ_j (i.e., no hold-and-wait)
  - **Step 1:** verify that a process matches its needs.
    if $REQ_j > Need_{ij}$ **abort** *–error, impossible*
  - **Step 2:** check if the requested amount is available.
    if $REQ_j > Avail_j$ **goto Step 1** *–Pi must wait*
  - **Step 3:** provisional allocation (i.e., guess and check).
    $Avail_j = Avail_j - REQ_j$
    $Hold_{ij} = Hold_{ij} + REQ_j$
    $Need_{ij} = Need_{ij} - REQ_j$
    if isSafe() then grant resources (system is safe) **else** cancel allocation; **goto Step 1***–Pi must wait*
- isSafe
  - Find out whether the system is in a safe state. **Work** and **Finish** are two temporary vectors.
  - **Step 1:** initialize.
    $Work_j = Avail_j$ for all j; $Finish_i = false$ for all i
  - **Step 2:** find a process P_i such that
    $Finish_i = false$ and $Need_{ij} \leq Work_j$, for all j
    if no such process, **goto Step 4**
  - **Step 3:** $Work_j = Work_j + Hold_{ij}$
    (i.e., pretend it finishes and frees up the resources)
    $Finish_i = true$ **goto Step 2**
  - **Step 4:** **if** $Finish_i = true$ for all i
    then return true*–yes, the system is safe*
    else return false*–no, the system is NOT safe*
- What is safe?
  - Safe with respect to some resource allocation
    - very safe
      $NEED_i \leq AVAIL$ for all Processes P_i. *Processes can run to completion in any order.*

  - safe (but take care)
    $NEED_i > AVAIL$ for some P_i
    $NEED_i \leq AVAIL$ for at least one P_i such that *There is at least one correct order in which the processes may complete their use of resources.*
  - unsafe (deadlock inevitable)
    $NEED_i > AVAIL$ for some P_i
    $NEED_i \leq AVAIL$ for at least one P_i *But some processes cannot complete successfully.*
  - deadlock
    $NEED_i > AVAIL$ for all P_i *Processes are already blocked or will become so as they request a resource.*

3.6 Deadlock Prevention

- The strategy of deadlock prevention is to design a system in such a way that the possibility of deadlock is excluded *a priori*
- Methods for preventing deadlock are of two classes:
  - *indirect methods* prevent the occurrence of one of the necessary conditions listed earlier.
  - *direct methods* prevent the occurrence of a circular wait condition.
- Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes
- Make it impossible that one of the four conditions for deadlock arise
- Mutual exclusion
  - In general, this condition cannot be disallowed
  - we can avoid assigning resources when not absolutely necessary
  - as few processes as possible should claim the resource
- Hold-and-wait
  - The hold and-wait condition can be prevented by requiring that a process request all its required resources at one time, and blocking the process until all requests can be granted simultaneously

  - Can we require processes to request all resources at once?
  - Most processes do not statically know about the resources they need
  - Wasteful, but works

- No preemption
  - One solution is that if a process holding certain resources is denied a further request, that process must release its unused resources and request them again, together with the additional resource
  - Preemption is feasible for some resources (e.g., processor and memory), but not for others (state must be saved and restored)

- Circular Wait
  - The circular wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering
  - order resources by an index:$R_1, R_2, \ldots$
  - requires that resources are always requested in order
  - P_1 holds R_i and requests R_j, and P_2 holds R_j and requests R_i is impossible
  - sometimes a feasible strategy, but not generally efficient

3.7 Summary of Deadlock strategies

Table 3.1: Summary of Deadlock strategies

Principle	***Resource Allocation Strategy***	***Different Schemes***	***Major Advantages***	***Major Disadvantages***
DETECTION	Very liberal; grant resources as requested	Invoke periodically to test for deadlock	**1-** Never delays process initiation **2-** Facilitates on-line handling	Inherent preemption losses
PREVENTION	Conservative; undercommits resources	Requesting all resources at once	**1-**Works well for processes with single burst of activity **2-** No preemption is needed	**1-** Inefficient **2-** Delays process initiation
		Preemption	Convenient when applied to resources whose state can be saved and restored easily	**1-** Preempts more often then necessary **2-**Subject to cyclic restart
		Resource ordering	**1-** Feasible to enforce via compile time checks **2-** Needs no run-time computation	**1-** Preempts without immediate use **2-** Disallows incremental resource requests
AVOIDANCE	Selects midway between that of detection and prevention	Manipulate to find at least one safe path	No preemption necessary	**1-** Future resource requirements must be known **2-** Processes can be blocked for long periods

Chapter 4

Memory Management

- The CPU fetches instructions and data of a program from memory; therefore, both the program and its data must reside in the main (RAM and ROM) memory
- What is memory Huge linear array of storage
- **malloc** library call
  - used to allocate and free memory
  - finds sufficient contiguous memory
  - reserves that memory
  - returns the address of the first byte of the memory
- **free** library call
  - give address of the first byte of memory to free
  - memory becomes available for reallocation
- both **malloc** and **free** are implemented using the **brk** system call
- Example of allocation (see the Fig. 4.1)

```
char *ptr=malloc(4096); //char* is address of a single byte
```

- Modern multiprogramming systems are capable of storing more than one program, together with the data they access, in the main memory
- A fundamental task of the **memory management** component of an operating system is to ensure safe execution of programs by providing:

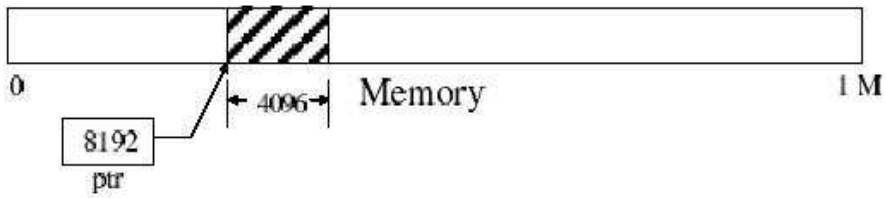


Figure 4.1: Allocating Memory.

  - Sharing of memory; issues are
    * *Transparency*; Several processes may co-exist, unaware of each other, in the main memory and run regardless of the number and location of processes.
    * *Efficiency*; CPU utilization must be preserved and memory must be fairly allocated. Want low overheads for memory management.
    * *Relocation* Ability of a program to run in different memory locations.
  - Memory protection; processes must not corrupt each other (nor the OS!)

- Information stored in main memory can be classified in a variety of ways:
  - Program (*code*) and data (*variables, constants*).
  - Read-only (*code, constants*) and read-write (*variables*).
  - Address (*e.g., pointers*) or data (*other variables*); binding (when memory is allocated for the object): static or dynamic
  - The compiler, linker, loader and run-time libraries all cooperate to manage this information.

- Before a program can be executed by the CPU, it must go through several steps:
  - **Compiling (translating)**–generates the object code.
  - **Linking**–combines the object code into a single self-sufficient *executable code*.
  - **Loading**–copies the executable code into memory. May include run-time linking with libraries.

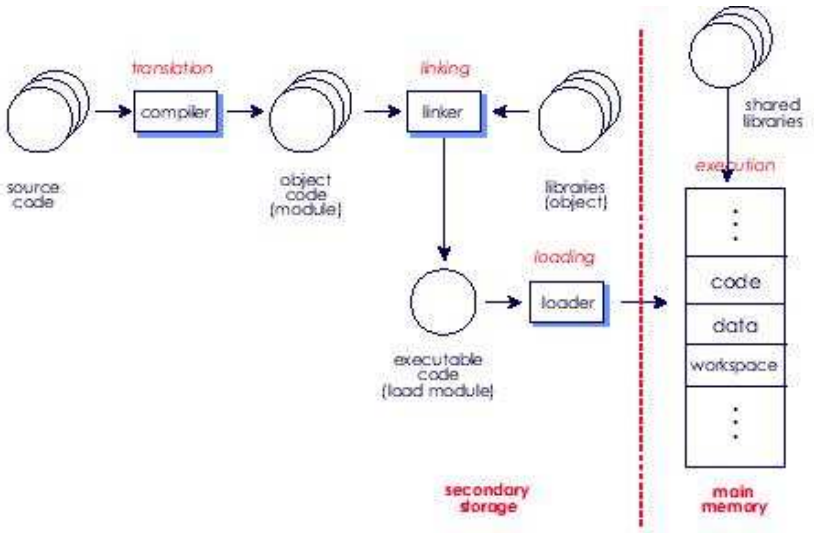


Figure 4.2: From source to executable code.

  - **Execution**–dynamic memory allocation.

- The process of associating program instructions and data (addresses) to physical memory addresses is called *address binding*, or *relocation*

  - **Static**–new locations are determined *before* execution
    - Compile time: The compiler or assembler translates *symbolic* addresses (e.g., variables) to *absolute* addresses.
    - Load time: The compiler translates symbolic addresses to *relative* (*relocatable*) addresses. The loader translates these to absolute addresses.
  - **Dynamic**–new locations are determined *during* execution
    - Run time: The program retains its relative addresses. The absolute addresses are generated by hardware.

4.1 Basic Memory Management

- An important task of a memory management system is to bring (load) programs into main memory for execution. The following *contiguous memory allocation* techniques were commonly employed by earlier operating systems:

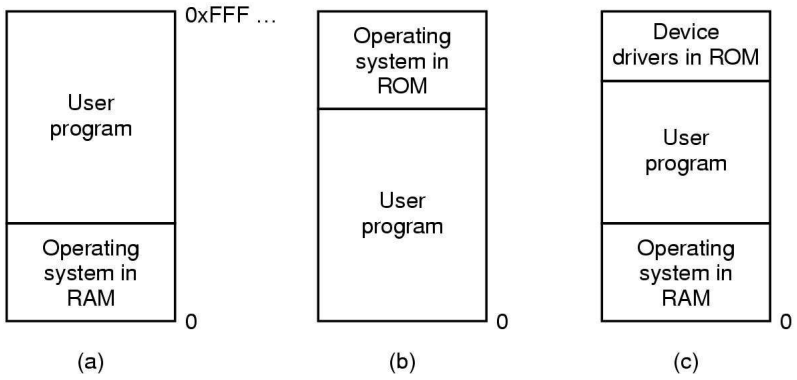


Figure 4.3: Three simple ways of organizing memory with an operating system and one user process.

  - Direct placement
  - Overlays
  - Partitioning
  - Techniques similar to those listed above are still used by some modern, dedicated special-purpose operating systems and real-time systems

- Memory management systems can be divided into two classes:

  1. those that move process back and forth between main memory and disk during execution (swapping and paging)
  2. those that do not

- starting with the second one

4.1.1 Monoprogramming without Swapping or Paging (see the Fig. 4.3)

- The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system

- Memory allocation is trivial. No special relocation is needed, because the user programs are always loaded (one at a time) into the same memory location (*absolute loading*). The linker produces the same loading address for every user program

- Examples:Early batch monitors, MS-DOS

4.1.2 Multiprogramming with Fixed Partitions (see the Fig. 4.4)

- Except on simple embedded systems, monoprogramming is hardly used any more
- A simple method to accommodate several programs in memory at the same time (to support multiprogramming) is partitioning
- The easiest way to achieve multiprogramming is simply to divide memory up into n (possibly unequal) partitions during system generation or startup
- When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it
- The aim of multiprogramming is to increase the CPU utilization
- $CPU\ utilization = 1 - p^n$, where n is the number of processes in the memory, p is waiting-time fraction that a process spends for I/O.
- say $p = 0.8$, means process spend 80 percent of their time waiting for I/O and n=10; then $CPU\ utilization = 89\%$, in other words CPU wasted 11 percent.

4.1.3 Relocation and Protection (see the Fig. 4.5)

- when a program is linked, the linker must know at what address the program will begin in the memory
- In order to provide basic protection among programs sharing the memory, partitioning techniques use a hardware capability known as *memory address mapping*, or address translation
- suppose that the first instruction is a call to a procedure at absolute address 100 within the binary file produced by the linker
- if this program is loaded in partition 1 (at address 100 K, see the Fig. 4.4), that instruction will jump to to absolute address 100, which is inside the operating system

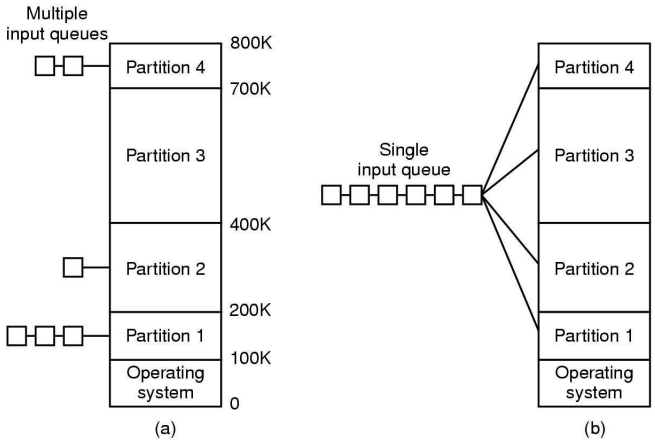


Figure 4.4: (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queues.

- what is needed is a call to 100K+100
- this problem is known as the relocation problem possible solution is is to modify the instructions as the program loaded into memory
- relocation during loading does not solve the protection problem
- A solution to both the relocation and protection problems is to equip the machine with two special hardware registers, called the **base** and **limits** registers

4.2 Swapping

- Two general approaches to memory management can be used, depending (in part) on the available hardware
- The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk
- The other strategy, called **virtual memory**, allows programs to run even they are only partially in main memory
- The basic idea of swapping is to treat main memory as a *preemptable* resource

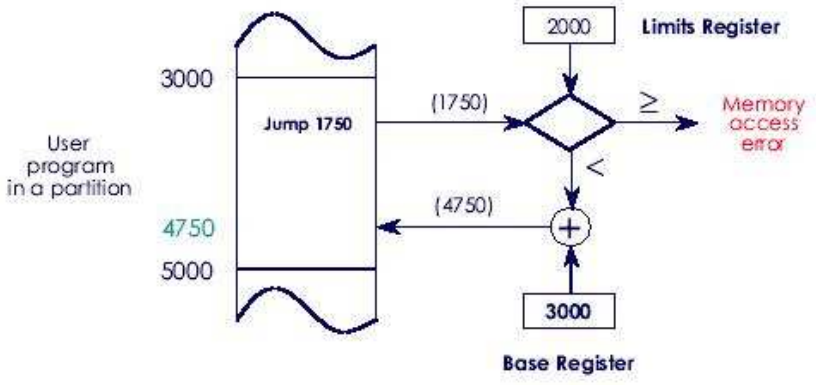


Figure 4.5: Address Translation.

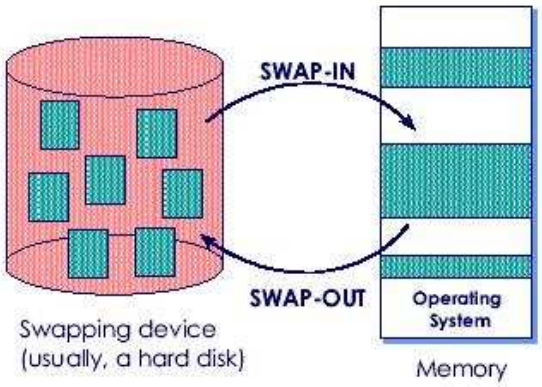


Figure 4.6: Swapping.

- A high-speed swapping device is used as the backing storage of the preempted processes
- *Fragmentation* refers to the unused memory that the memory management system cannot allocate
  - *Internal fragmentation*; Waste of memory *within* a partition, caused by the difference between the size of a partition and the process loaded. Severe in static partitioning schemes (Multiprogramming with Fixed Partitions (MFT)).
  - *External fragmentation*; Waste of memory *between* partitions, caused by scattered noncontiguous free space. Severe in dynamic parti-

tioning schemes (Multiprogramming with Variable Partitions (MVT), swapping).

- *Compaction* (aka relocation) is a technique that is used to overcome external fragmentation
- The responsibilities of a swapper include:
  - Selection of processes to swap out *criteria*: suspended/blocked state, low priority, time spent in memory
  - Selection of processes to swap in *criteria*: time spent on swapping device, priority
  - Allocation and management of swap space on a swapping device. Swap space can be:
    - system wide
    - dedicated (e.g., swap partition or disk)

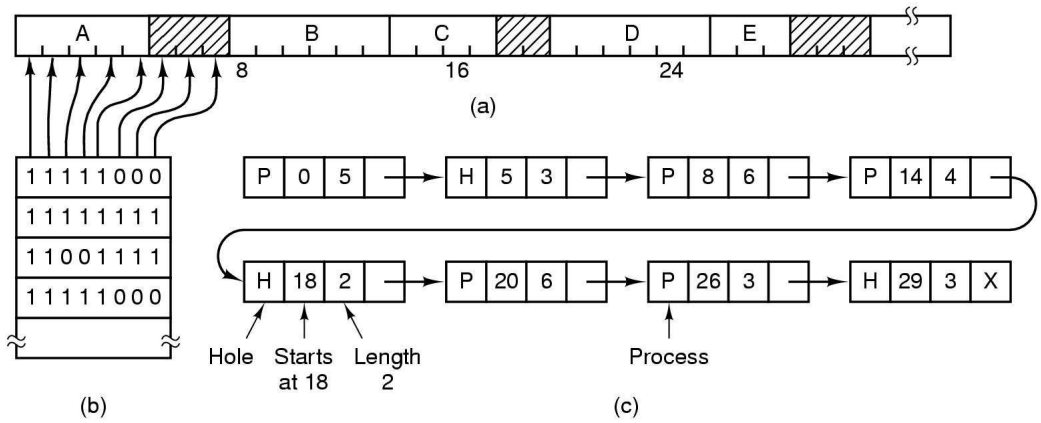


Figure 4.7: (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded region (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

4.2.1 Memory Management cont.

- Memory allocation and freeing operations are partially predictable. Since the organization is hierarchical, the freeing operates in reverse (opposite) order (Stack Organization)
- Allocation and release of heap space is totally random. Heaps are used for allocation of arbitrary list structures and complex data organizations. As programs execute (and allocate and free structures), heap space will fill with holes (unallocated space, i.e., fragmentation) (Heap Organization)
- How do we know *when* memory can be freed? It is trivial when a memory block is used by one process
- However, this task becomes difficult when a block is shared (e.g., accessed through pointers) by several processes
- Two problems with reclaiming memory:
  - **Dangling pointers**: occur when the original allocator *frees* a shared pointer.
  - **Memory leaks**: occur when we forget to free storage, even when it will not or cannot be used again. This is a common and serious

problem. Unacceptable in an OS.

- Memory Management with Bitmaps (see Fig. 4.7)
  This technique divides memory into fixed-size blocks (e.g., sectors of 256-byte blocks) and keeps an array of bits (bit map), one bit for each block

- Memory Management with Linked Lists (see Fig. 4.7)
  A *free list* keeps track of the unused memory. There are several algorithms that can be used, depending on the way the unused memory blocks are allocated (Dynamic Partitioning Placement Algorithm):

  - **First-fit**: Allocate *first* hole that is big enough
    - Scan the list for the first entry that fits
    - If greater in size, break it into an allocated and free part
    - May have many processes loaded in the front end of memory that must be searched over when trying to find a free block
    - May have lots of unusable holes at the beginning.
    - External fragmentation
  - **Next-fit**
    - Like first-fit, except it begins its search from the point in list where the last request succeeded instead of at the beginning.
    - More often allocates a block of memory at the end of memory where the largest block is found
    - The largest block of memory is broken up into smaller blocks
    - Compaction is required to obtain a large block at the end of memory
    - Simulations show it is slightly slower
  - **Best-fit**: Allocate *smallest* hole that is big enough;
    - Chooses the block that is closest in size to the request
    - Poor performer
    - Has to search complete list, unless ordered by size.
    - Since smallest block is chosen for a process, the smallest amount of fragmentation is left memory compaction must be done more often
  - **Worst-fit**: Allocate *largest* hole;
    - Chooses the block that is largest in size (worst-fit)

    * Idea is to leave a usable fragment left over
    * Poor performer
    * must also search entire list to find largest leftover hole (keep list in size order)
    * Simulations show it is not a good idea

4.3 Virtual Memory

- Differentiation of user logical memory from physical memory
  - only part of program needs to be in memory for execution
  - logical address space can be >> than physical address space
  - allows address spaces to be shared by processes
  - allows more efficient process creation
- Virtual memory (VM) can be implemented via:
  - demand paging
  - demand segmentation

4.3.1 Paging (see the Fig. 4.8)

- A VM Larger Than Physical Memory
- Logical address space of a process can be noncontiguous; process allocated physical memory wherever latter available
- Bring page into memory only when needed
  - less I/O needed
  - less memory needed
  - faster response
  - more processes
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - Not in memory ⇒ bring to memory
- Reference may result from:

– instruction fetch

– data reference

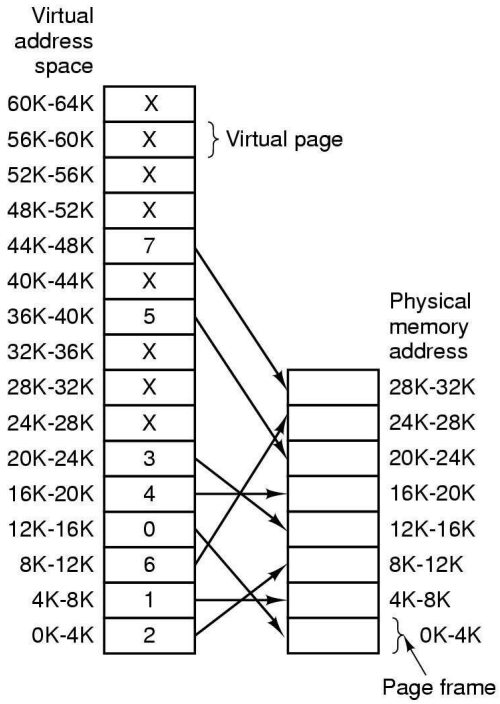


Figure 4.8: The relation between virtual address and physical memory addresses is given by a page table.

- Divide physical memory into fixed-sized blocks called **frames** (size power of 2, typically 512 bytes - 8KB)
- Divide logical memory into blocks of same size called **pages**
- A **Present/Absent** bit keeps track of which pages are physically present in memory
- A page table defines (maps) the base address of pages for each frame in the main memory
- The major goals of paging are to make memory allocation and swapping easier and to reduce fragmentation
- Paging also allows allocation of non-contiguous memory (i.e., pages need not be adjacent.)

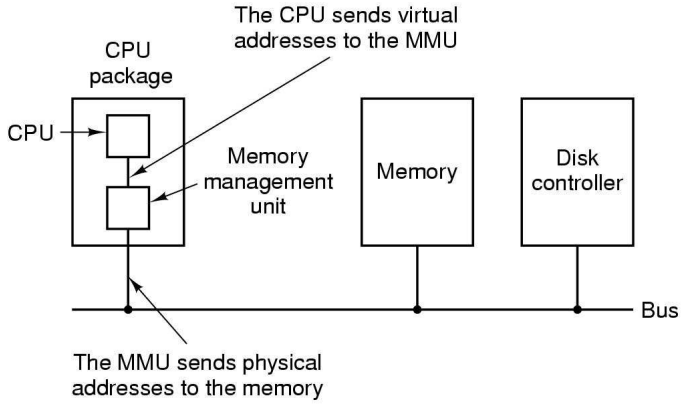


Figure 4.9: The position and function of the MMU.

- Keep track of all free frames
- Set up page table to translate logical to physical addresses
- Internal fragmentation, sometimes less than full page needed
- Dynamic relocation, each program-generated address (logical address) is translated to hardware address (physical address) at runtime for every reference, by a hardware device known as the memory management unit (MMU)
- Memory-Management Unit
  - Hardware maps logical to physical address
  - With MMU, value in relocation register added to every address generated by user process when sent to memory
  - User program deals with *logical* addresses; never sees *real* physical addresses
- Address Translation Scheme; address generated by CPU divided into:
  - *Page number (p)* used to index into *page table* with base address of each page in physical memory
  - *Page offset (d)* combined with base address defines physical address for memory system

- What happens when an executing program references an address that is not in main memory? (see Fig. 4.10)
- The page table is extended with an extra bit, present/absent bit
- **Page Fault:** Access to a page whose present bit is not set causes a special hardware trap, called page fault
- Initially, all the present bits are cleared. While doing the address translation, the MMU checks to see if this bit is set.
  1. Trap to OS
  2. Save user registers and process state
  3. Determine that interrupt was page fault
  4. Check that page reference was legal, determine location of page on disk
  5. Issue read from disk to free (physical) frame:
     - Wait in queue for device to service read request
     - Wait for device seek / rotational latency
     - Wait for transfer
  6. While waiting for disk: allocate CPU to another process
  7. Interrupt from disk
  8. Save registers and process state for other process
  9. Determine that interrupt was from disk
  10. Update page / OS tables to show page is in memory
  11. Wait for CPU to be allocated to this process
  12. Restore registers, process state, page table
  13. Restart instruction
- When a page fault occurs the operating system brings the page into memory, sets the corresponding present bit, and restarts the execution of the instruction
- What happens if no free frame?
  - Page replacement find some page in memory, ideally not in use, swap it out
    - algorithm performance

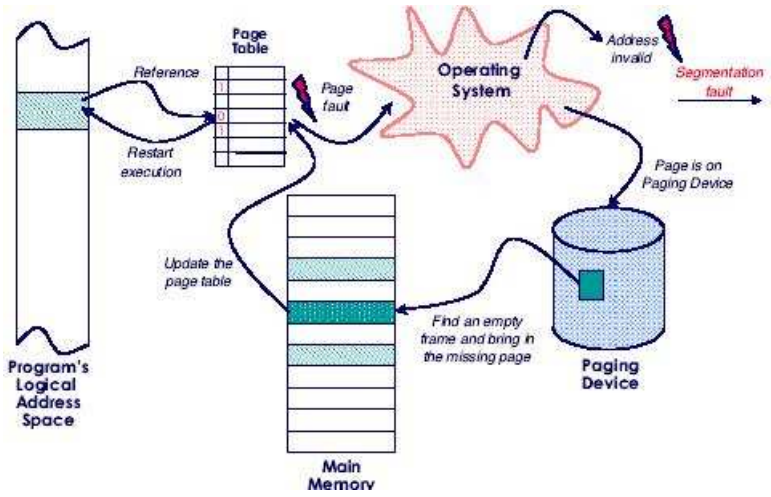


Figure 4.10: Page fault handling by picture.

    - want to minimize number of page faults
  - Same page may be brought into memory several times

- Problem: Both paging and segmentation (we will discuss later) schemes introduce extra memory references to access translation tables.
- Solution? Translation buffers (like caches)
- Based on the notion of **locality** (at a given time a process is only using a few pages or segments), a very fast but small associative (content addressable) memory is used to store a few of the translation table entries
- This memory is known as a translation look-aside buffer or TLB
- Similar to storing memory addresses in TLBs, frequently used data in main memory can also be stored in fast buffers, called cache memory, or simply cache (see Fig. 4.11)
- Basically, memory access occurs as follows:

```
for each memory reference
 if data is not in cache <miss>
  if cache is full
   remove some data
  if read access
```

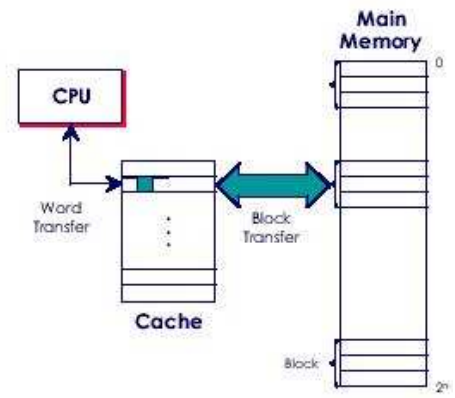


Figure 4.11: Memory Caching.

```
  issue memory read
  place data in cache
  return data
else <hit>
 if read access
  return data
 else
  store data in cache
```

- The idea is to make frequent memory accesses faster!
- Cache terminology;
  - **Cache hit**: item is in the cache.
  - **Cache miss**: item is not in the cache; must do a full operation. **Categories of cache miss**:
    - *Compulsory*: the first reference will always miss.
    - *Capacity*: non-compulsory misses because of limited cache size.
  - **Effective access time**: P(hit) * cost of hit + P(miss)* cost of miss, where P(hit) = 1 - P(miss)

4.3.2 Page Tables (see Fig. 4.12)

- Page table kept in main memory

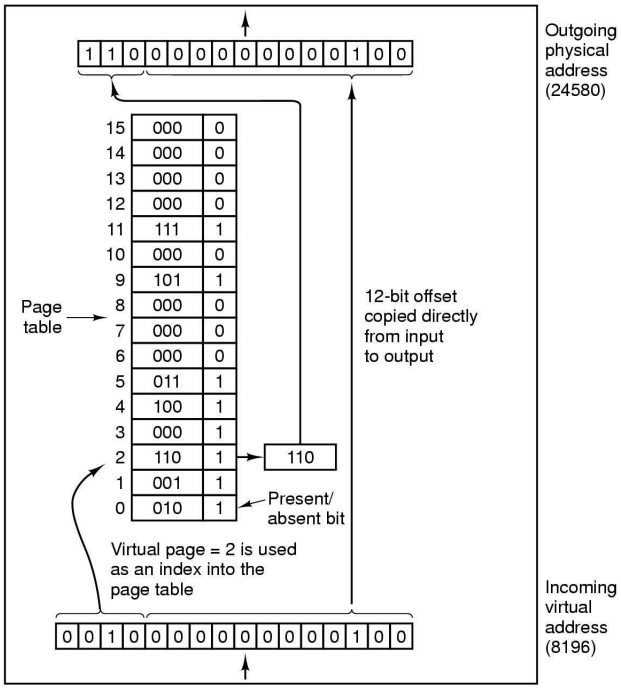


Figure 4.12: The internal operation of the MMU with 16 4-KB pages.

- The Modified and Referenced bits keep track of the page usage. If the page in it has been modified, it must be written back to the disk. This bit s sometimes called the **dirty bit.**
- *Page-table base register* (PTBR) points to page table
- *Page-table length register* (PRLR) has size of page table
- In this scheme every data/instruction access needs 2 memory accesses: page table and data/instruction
- 2-memory access problem can be solved by special fast-lookup hardware cache using *associative memory* called *translation look-aside buffer (TLB)*
- Address space may be very large, e.g.:
  - 32-bit addresses 4GB
  - 64-bit addresses millions of TB
- May have big gaps (*sparse*), e.g.

  - stack grows down from high memory
  - dynamic allocation grows up from low memory

- Page table very large, big waste of memory

4.3.3 Inverted Page Tables

- The inverted page table has one entry for each memory frame.
- Entry is virtual address of page stored in real location, with information about owning process
- Decreases memory to store page table, increases time to search for page translation
- Hashing is used to speedup table search. Hash table limits search to O(1) entries
- Popular with virtual space >> physical
- Hard to handle *aliases* (> 1 virtual page maps to 1 physical page)
- The inverted page table can either be per process or system-wide. In the latter case, a new entry, PID (process id)is added to the table.
- Adv: independent of size of address space; small table(s) if we have large logical address spaces.

4.3.4 Basic policies

- The hardware only provides the basic capabilities for virtual memory. The operating system, on the other hand, must make several decisions:
  - *Allocation*–how much real memory to allocate to each (*ready)* program?
    - In general, the allocation policy deals with conflicting requirements:
      - The fewer the frames allocated for a program, the higher the page fault rate
      - The fewer the frames allocated for a program, the more programs can reside in memory; thus, decreasing the need of swapping.

        - Allocating additional frames to a program beyond a certain number results in little or only moderate gain in performance.
    * The number of allocated pages (also known as *resident set size*) can be *fixed* or can be *variable* during the execution of a program.

- *Fetching*–when to bring the pages into main memory?
    * **Demand paging**; Start a program with no pages loaded; wait until it references a page; then load the page (this is the most common approach used in paging systems.)
    * **Request paging**; Similar to overlays, let the user identify which pages are needed (not practical, leads to over estimation and also user may not know what to ask for.)
    * **Pre-paging**; Start with one or a few pages preloaded. As pages are referenced, bring in other (not yet referenced) pages too.
    * Opposite to fetching, the *cleaning policy* deals with determining when a modified (dirty) page should be written back to the paging device.

- *Placement*–where in the memory the fetched page should be loaded?
    * This policy usually follows the rules about paging and segmentation.
    * Given the matching sizes of a page and a frame, placement with paging is straightforward.
    * Segmentation requires more careful placement, especially when *not* combined with paging. Placement in pure segmentation is an important issue and *must* consider free memory management policies.
    * With the recent developments in *non-uniform memory access (NUMA)* distributed memory multiprocessor systems, placement does become a major concern.

- *Replacement* (see the Fig. 4.13) –what page should be removed from main memory?
    * The most studied area of the memory management is the replacement policy or victim selection to satisfy a page fault.

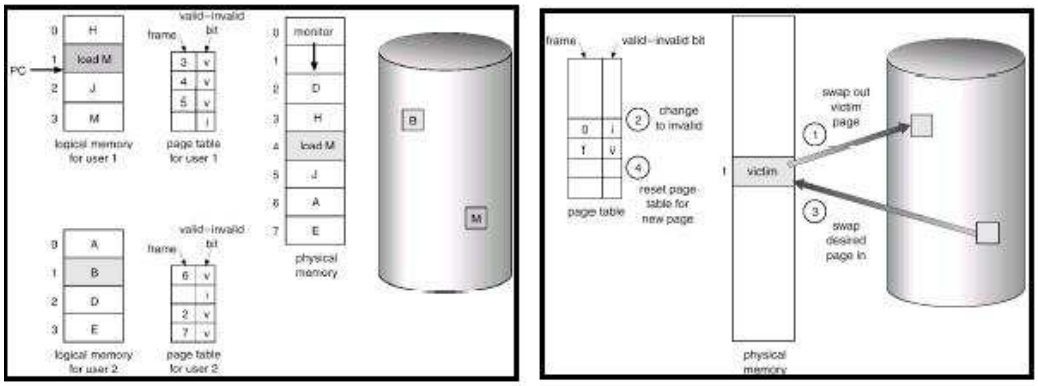


Figure 4.13: Page Replacement.

4.3.5 Page Replacement Algorithms

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use *dirty (modify)* bit to reduce overhead of page transfers; only modified pages written back to disk
- Page replacement completes separation between logical memory and physical memory; large virtual memory can be provided on smaller physical memory
- Find location of desired page on disk
- Find free frame:
  - if free frame, use it.
  - if no free frame, page replacement algorithm selects *victim* frame
  - If victim *dirty* write back to disk
- Read desired page into (newly) free frame; update page and frame tables
- Restart faulting process
- Want lowest page-fault rate
- Evaluate algorithm by running on given string of memory references (reference string) and compute number of page faults

Table 4.1: Page replacement algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU	Very crude
FIFO	Might throw out important pages
Second Chance	Big improvement over FIFO
Clock	Realistic
LRU	Excellent, but difficult to implement exactly
NFU	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

- The Optimal Page Replacement Algorithm; the page that will not be referenced again for the longest time is replaced (prediction of the future; purely theoretical, but useful for comparison.)
- The Not Recently Used Page Replacement Algorithm; algorithm removes a page at random
- The First-In, First-Out (FIFO) Page Replacement Algorithm; FIFO the frames are treated as a circular list; the oldest (longest resident) page is replaced
- The Second Chance Page Replacement Algorithm; look for an old page that has not been referenced in the previous clock interval, avoids the problem of throwing out of heavily used page
- The Least Recently Used (LRU) Page Replacement Algorithm; LRU the frame whose contents have not been used for the longest time is replaced
- Summary of Page Replacement Algorithms

4.3.6 Page Replacement Cont.

- Global vs. Local Allocation
  - **Global** replacement; process selects replacement frame from set of all frames; process can take frame from another
  - **Local** replacement each process selects only from its own set of allocated frames

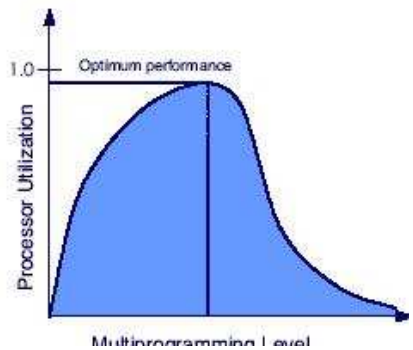


Figure 4.14: Trashing.

- **Frame locking**; frames that belong to resident kernel, or are used for critical purposes, may be locked for improved performance.
- **Page buffering**; victim frames are grouped into two categories: those that hold unmodified (clean) pages and modified (dirty) pages
- **Thrashing (see Fig. 4.14)**;
  - The number of processes that are in the memory determines the multiprogramming (MP) level.
  - The effectiveness of virtual memory management is closely related to the MP level.
  - When there are just a few processes in memory, the possibility of processes being blocked and thus swapped out is higher.
  - When there are far too many processes (i.e., memory is overcommitted), the resident set of each process is smaller.
  - This leads to higher page fault frequency, causing the system to exhibit a behavior known as thrashing.
  - In other words, the system is spending its time moving pages in and out of memory and hardly doing anything useful.
  - process spends more time paging than executing
  - The only way to eliminate thrashing is to reduce the multiprogramming level by suspending one or more process(es).
  - Victim process(es) can be the: lowest priority process, faulting process, newest process, process with the smallest resident set, process with the largest resident set

– Student analogy to thrashing: Too many courses!

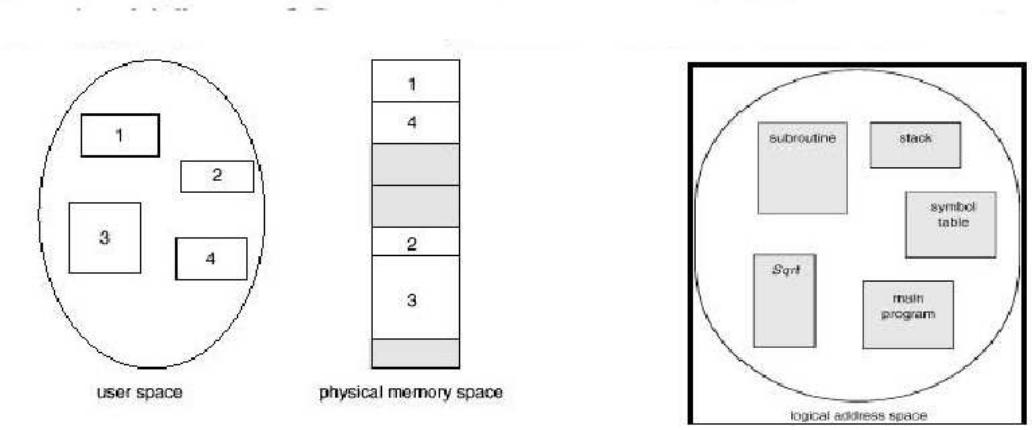


Figure 4.15: Logical View of Segmentation (left) , User's View of a Program (right).

4.4 Segmentation

- The most important problem with *base-and-limits* (see Fig. 4.5) relocation is that there is only one segment for each process
- Segmentation generalizes the base-and-limits technique by allowing each process to be split over several segments (i.e., multiple base-limits pairs)
- *Segment table* maps 2-dimensional physical addresses (segment-number, offset); each table entry has:
  - base; contains starting physical address where segments reside in memory
  - *limit* specifies length of segment
- Table entries are filled as new segments are allocated for the process
- A segment is a region of contiguous memory. Although the segments may be scattered in memory, each segment is mapped to a contiguous region
- Memory-management scheme that supports user view of memory (see Fig. 4.15)
- Program is collection of segments. Segment a logical unit such as: main program, procedure, function, method, object, local variables,

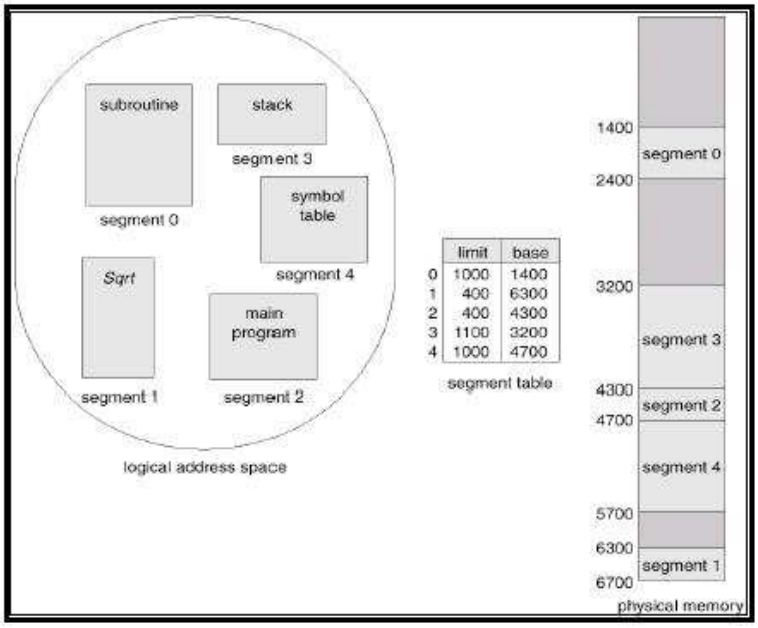


Figure 4.16: Example of Segmentation

global variables, common block, stack, symbol table, array (see Fig. 4.16)

- When a process is created, an empty segment table is inserted into the process control block (PCB)
- The segments are returned to the free segment pool when the process terminates
- Segmentation, as well as the base and limits approach, causes external fragmentation (because they require contiguous physical memory) and requires memory compaction
- An advantage of the approach is that only a segment, instead of a whole process, may be swapped to make room for the (new) process.
- Like paging, use virtual addresses and use disk to make memory look bigger that it really is
- Segmentation can be implemented with or without paging
- *Segment-table base register (STBR)* points to segment table's location in memory

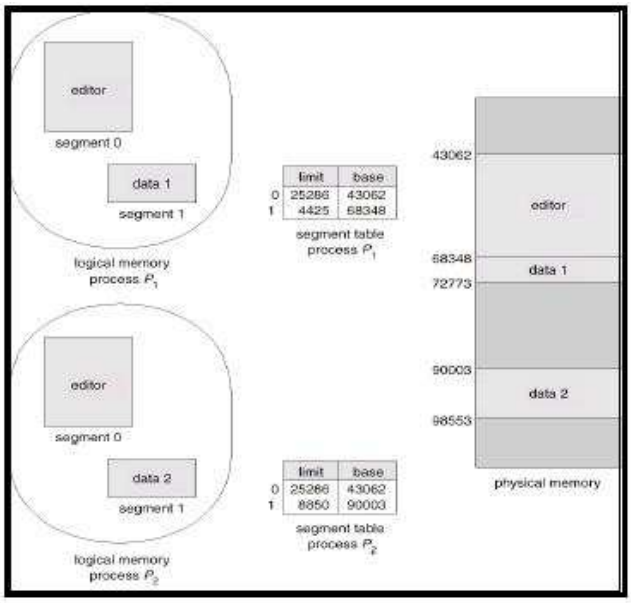


Figure 4.17: Sharing of Segmentation

- *Segment-table length register (STLR)* indicates number of segments used by a program , segment number s is legal if $s <$ STLR
- Segmentation Architecture
  - Relocation; dynamic, by segment table
  - Sharing; shared segments, same segment number
  - Allocation; first fit/best fit, external fragmentation
  - Protection: with each entry in segment table: illegal segment, read/write/execute privileges
  - Protection bits associated with segments; code sharing at segment level
  - Since segments vary in length, memory allocation a dynamic storage-allocation problem

4.4.1 Segmentation with Paging

- Advantages of Segmentation
  - Different protection for different segments read-only status for code

  - Enables sharing of selected segments (see Fig. 4.17)
  - Easier to relocate segments than entire address space
  - Enables sparse allocation of address space

- Disadvantages of Segmentation
  - Still expensive/difficult to allocate contiguous memory to segments
  - External fragmentation: Wasted memory
  - Paging; Allocation easier, Reduces fragmentation

- Advantages of Paging
  - Fast to allocate and free;
    * Alloc: Keep free list of free pages and grab first page in list, no searching by first-fit, best-fit
    * Free: Add page to free list, no inserting by address or size
  - Easy to swap-out memory to disk
    * Page size matches disk block size
    * Can swap-out only necessary pages
    * Easy to swap-in pages back from disk

- Disadvantages of Paging
  - Additional memory reference: Inefficient. Page table too large to store as registers in MMU. Page tables kept in main memory. MMU stores only base address of page table.
  - Storage for page tables may be substantial
    * Simple page table: Require entry for all pages in address space. Even if actual pages are not allocated
    * Partial solution: Base and bounds (limits) for page table. Only processes with large address spaces need large page tables. Does not help processes with stack at top and heap at bottom.
  - Internal fragmentation: Page size does not match allocation size
    * How much memory is wasted (on average) per process?
    * Wasted memory grows with larger pages

- Combine Paging and Segmentation

  - Structure
    - Segments correspond to logical units: code, data, stack. Segments vary in size and are often large
    - Each segment contains one or more (fixed-size) pages
  - Two levels of mapping to make tables manageable (2 look-ups!)
    - Page table for each segment
    - Base (real address) and bound (size) for each page table

- Segments + Pages Advantages
  - Advantages of Segments
    - Supports sparse address spaces. If segment is not used, no need for page table. Decreases memory required for page tables.
  - Advantages of Paging
    - Eliminate external fragmentation
    - Segments to grow without any reshuffling
  - Advantages of Both. Increases flexibility of sharing. Share at two levels: Page or segment (entire page table)

- Segments + Pages Disadvantages
  - Internal fragmentation increases. Last page of every segment in every process
  - Increases overhead of accessing memory
    - Translation tables in main memory
    - 1 or 2 overhead references for every real reference
  - Large page tables
    - Do not want to allocate page tables contiguously
    - More problematic with more logical address bits
    - Two potential solutions: Page the user page tables (multilevel page table), Inverted page table

4.4.2 Segmentation with Paging: MULTICS

- MULTICS solved problems of external fragmentation and lengthy search times by paging segments

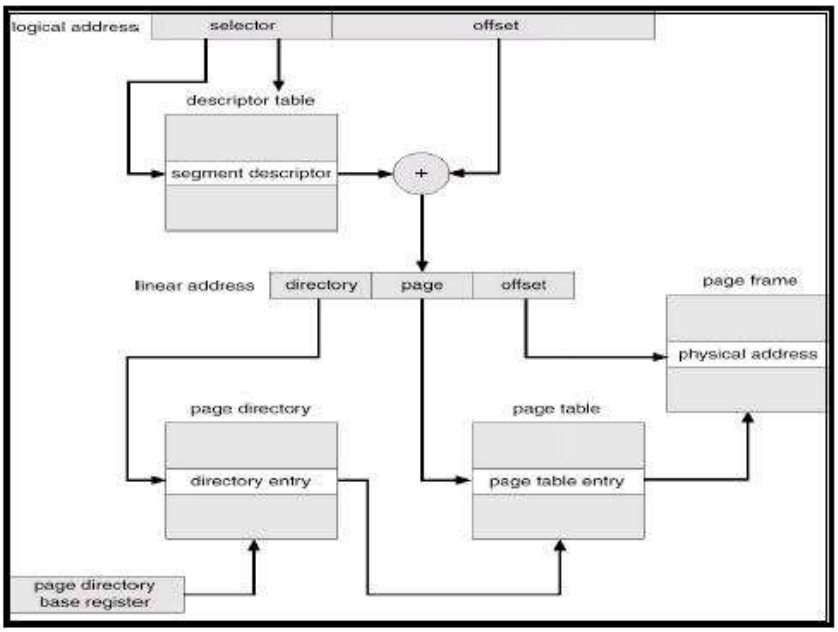


Figure 4.18: Intel 386 Address Translation

- Solution differs from pure segmentation: segment-table entry contains not base address of segment, but base address of *page table* for this segment

4.4.3 Segmentation with Paging: The Intel Pentium (see Fig. 4.18)

- Intel 386 and later use segmentation with paging
- OS/2 uses full scheme
- Other OSes mostly only use pages; Linux, Windows NT and successors

Chapter 5

INPUT/OUTPUT

5.1 Principles of I/O Hardware

- There exists a large variety of I/O devices:
  - Many of the with different properties
  - They seem to require different interfaces to manipulate and manage them
- We don't want a new interface for every device
- Diverse, but similar interfaces leads to code duplication
- Challenge: Uniform and efficient approach to I/O
- Common concepts
  - port
  - bus
  - controller (host adapter)
- each port is given a special address (see Table 5.1)
- **communication**, use an assembly instruction (high–level languages only work with main memory) to read/write a port; e.g., OUT *port, reg*: writes the value in CPU register *reg* to I/O port *port*
- **protection**, users should have access to some I/O devices but not to others
- I/O instructions control devices

Table 5.1: Device I/O Port Locations on PCs (Partial).

I/O address range (hexadecimal)	Device
000-00F	DMA Controller
020-021	Interrupt Controller
040-043	Timer
200-20F	Game Controller
2F8-2FF	Serial port (secondary)
320-32F	Hard disk Controller
378-37F	Parallel port
3D0-3DF	Graphics Controller
3F0-3F7	Diskette drive Controller
3F8-3FF	Serial port (primary)

- Devices have addresses, used by
  - direct I/O instructions
  - memory-mapped I/O

5.1.1 Device Controllers (see Fig. 5.1)

- I/O devices have controllers; disk controller, monitor controller, etc.
- controller manipulates/interprets electrical signals to/from the device
- controller accepts commands from CPU or provides data to CPU
- controller and CPU communicate over I/O ports; control, status, input and output registers

5.1.2 I/O Devices

- Categories of I/O Devices (by usage)
  - Human readable
    - Used to communicate with the user
    - Printers, Video Display, Keyboard, Mouse
  - Machine readable
    - Used to communicate with electronic equipment
    - Disk and tape drives, Sensors, Controllers, Actuators

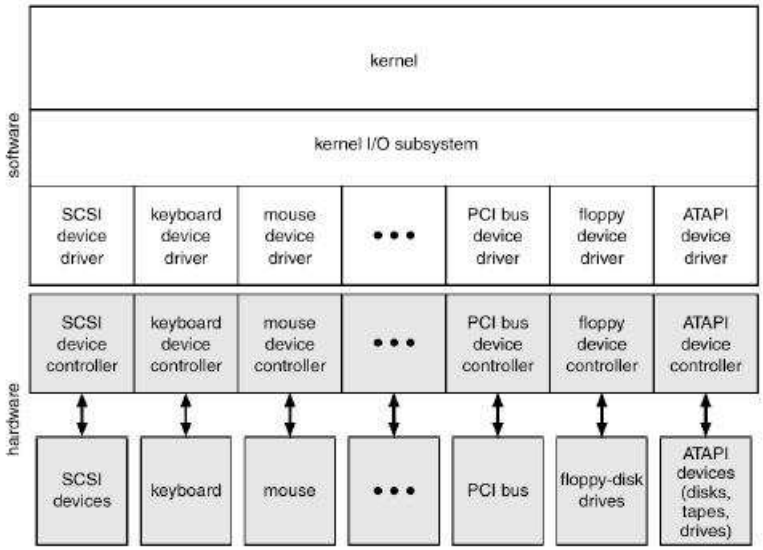


Figure 5.1: A kernel I/O structure.

  - Communication
    - Used to communicate with remote devices
    - Ethernet, Modems, Wireless

- I/O system calls abstract device behaviors in generic classes (see Fig. 5.1)
- Device-driver layer hides I/O-controller differences from kernel
- Devices vary in many dimensions
  - character-stream or block
  - sequential or random-access
  - sharable or dedicated
  - speed of operation
  - read-write, read only, or write only
- Block and Character Devices; Block devices include disk drive
  - commands include read, write, seek
  - raw I/O or file-system access

  - file system maps location i onto block + offset
  - memory-mapped file access possible

- Character devices include keyboard, mouse, serial port
  - commands include get, put
  - libraries layered on top allow line editing

5.1.3 Characteristics (see Table 5.2) and Differences in I/O Devices

Table 5.2: Characteristics of I/O Devices

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

- Application
  - Disk used to store files requires file management software
  - Disk used to store virtual memory pages needs special hardware and software to support it
  - Terminal used by system administrator may have a higher priority
- Complexity of control;
  - Unit of transfer; Data may be transferred as a stream of bytes for a terminal or in larger blocks for a disk
  - Data representation; Encoding schemes
  - Error conditions; Devices respond to errors differently
- Blocking; process suspended until I/O completed

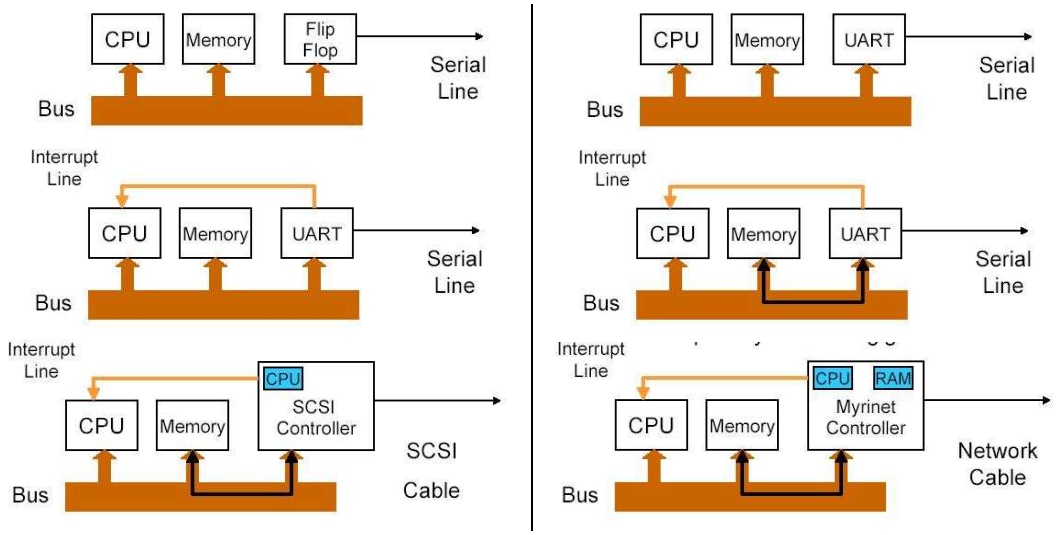


Figure 5.2: Evolution of the I/O Function

  - Easy to use and understand
  - Insufficient for some needs

- Nonblocking; I/O call returns as much as available
  - user interface, data copy (buffered I/O)
  - implemented via multi-threading code for I/O call
  - returns quickly with count of bytes transferred

- Asynchronous; process runs while I/O executes
  - difficult to use
  - I/O subsystem signals process when I/O completed, e.g., call-backs: pointer to completion code

5.1.4 Evolution of the I/O Function (see Fig. 5.2)

- Processor directly controls a peripheral device. Example: CPU controls a flip-flop to implement a serial line

- Controller or I/O module is added
  - Processor uses programmed I/O without interrupts
  - Processor does not need to handle details of external devices

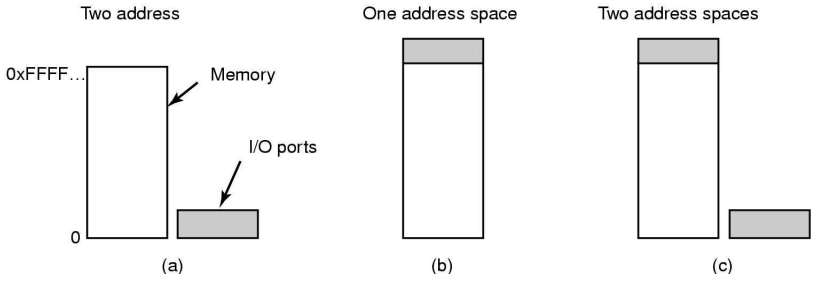


Figure 5.3: a) Separate I/O and memory space. b) Memory-mapped I/O. c) Hybrid.

  - Example: A Universal Asynchronous Receiver Transmitter
    - CPU simply reads and writes bytes to I/O controller
    - I/O controller responsible for managing the signalling

- Controller or I/O module with interrupts. Processor does not spend time waiting for an I/O operation to be performed
- Direct Memory Access
  - Blocks of data are moved into memory without involving the processor
  - Processor involved at beginning and end only
- I/O module has a separate processor. Example: SCSI controller, controller CPU executes SCSI program code out of main memory
- I/O processor
  - I/O module has its own local memory, internal bus, etc.
  - It is a computer in its own right.
  - Example: Myrinet Multi-gigabit Network Controller

5.1.5 Memory-Mapped I/O (see Fig. 5.3)

- Separate I/O and memory space
  - I/O controller registers appear as I/O ports
  - Accessed with special I/O instructions

- Memory-mapped I/O
  - Controller registers appear as memory
  - Use normal load/store instructions to access
- Hybrid; x86 has both ports and memory mapped I/O
- Bus Architectures (see Fig. 5.4)

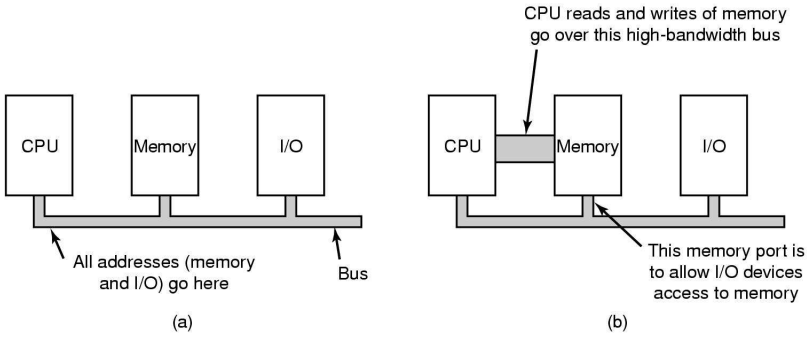


Figure 5.4: a) A single-bus architecture. b) A dual-bus memory architecture.

  - A single-bus architecture; if the computer has a single bus, having everyone look at every address is straightforward
  - A dual-bus memory architecture; the trend in modern personal computers is to have a dedicated high–speed memory bus. This bus is tailored for optimize memory performance, with no compromises for the sake of slow I/O devices. Pentium systems even have three external buses (memory, PCI, ISA)

5.1.6 Direct Memory Access (DMA)

- Takes control of the bus from the CPU to transfer data to and from memory over the system bus
- Cycle stealing is used to transfer data on the system bus
- The instruction cycle is suspended so data can be transferred
- The CPU pauses one bus cycle, CPU Cache can hopefully avoid such pauses
- Reduced number of interrupts occur, No expensive context switches

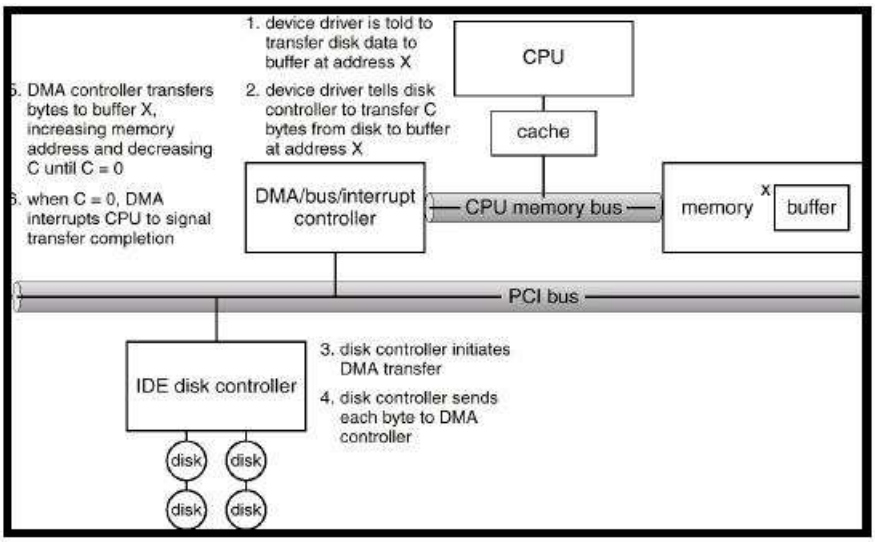


Figure 5.5: The Process to Perform DMA Transfer.

- Cycle stealing causes the CPU to execute more slowly; Still more efficient than CPU doing transfer itself
- The CPU cache can hide some bus transactions
- Number of required busy cycles can be cut by
  - integrating the DMA and I/O functions
  - Path between DMA module and I/O module that does not include the system bus
- The Process to Perform DMA Transfer (see Fig. 5.5)
  1. device driver is told to transfer disk data to buffer at address X
  2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
  3. disk controller initiates DMA transfer
  4. disk controller sends each byte to DMA controller
  5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C=0
  6. when C=0, DMA interrupts CPU to signal transfer completion

5.1.7 Interrupts Revisited (see Fig. 5.6)

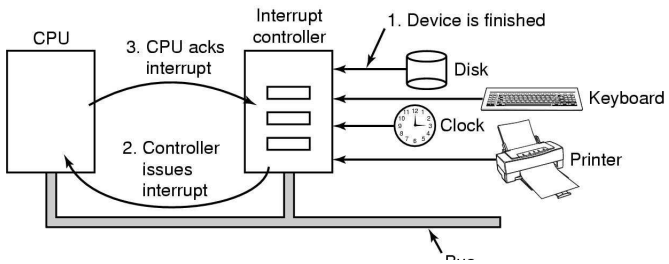


Figure 5.6: How interrupts happen. The connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires.

- CPU interrupt request line triggered by I/O device
- Interrupt handler receives interrupts
- Maskable to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler based on priority some unmaskable
- Interrupt mechanism also used for exceptions, traps

5.2 Principles of I/O Software

5.2.1 Programmed I/O (see Fig. 5.7a)

- Also called *polling*, or *busy waiting*
- I/O module (controller) performs the action, not the processor
- Sets appropriate bits in the I/O status register
- No interrupts occur
- Processor checks status until operation is complete; Wastes CPU cycles

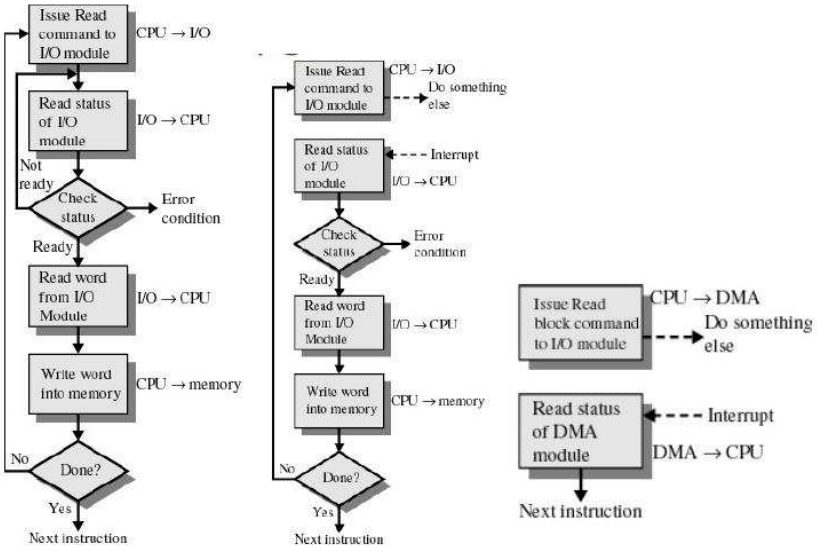


Figure 5.7: a) Programmed I/O. b) Interrupt-Driven I/O. c) Direct Memory Access.

5.2.2 Interrupt-Driven I/O (see Fig. 5.7b)

- Processor is interrupted when I/O module (controller) ready to exchange data
- Processor is free to do other work
- No needless waiting
- Consumes a lot of processor time because every word read or written passes through the processor

5.2.3 Direct Memory Access (see Fig. 5.7c)

- Transfers a block of data directly to or from memory
- An interrupt is sent when the task is complete
- The processor is only involved at the beginning and end of the transfer

5.3 Operating System Design Issues

- Efficiency
  - Most I/O devices slow compared to main memory (and the CPU)
    - Use of multiprogramming allows for some processes to be waiting on I/O while another process executes
    - Often I/O still cannot keep up with processor speed
    - Swapping may used to bring in additional Ready processes; More I/O operations
- **Optimise I/O efficiency especially Disk & Network I/O**
- The quest for generality/uniformity:
  - Ideally, handle all I/O devices in the same way; Both in the OS and in user applications
  - Problem:
    - Diversity of I/O devices
    - Especially, different access methods (random access versus stream based) as well as vastly different data rates.
    - Generality often compromises efficiency!
  - Hide most of the details of device I/O in lower-level routines so that processes and upper levels see devices in general terms such as read, write, open, close, lock, unlock

5.4 I/O Software Layers (see Fig. 5.8)

5.4.1 Interrupt Handlers

- Interrupt handlers are best hidden
  - Can execute at almost any time
  - Raise (complex) concurrency issues in the kernel
  - Have similar problems within applications if interrupts are propagated to user-level code (via signals, upcalls).
  - Generally, have driver starting an I/O operation block until interrupt notifies of completion; Example dev_read() waits on semaphore that the interrupt handler signals

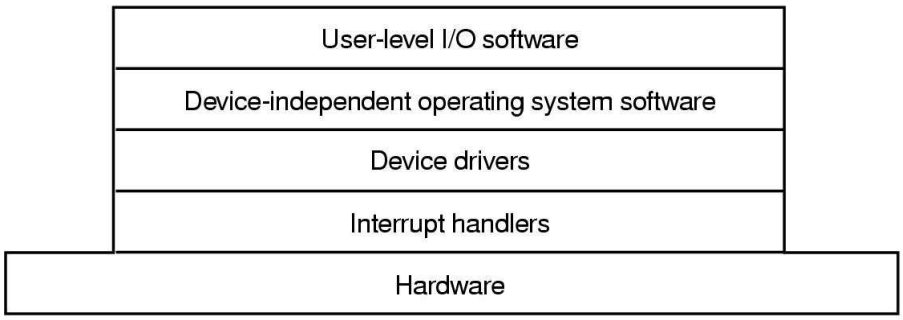


Figure 5.8: Layers of the I/O Software System.

- Interrupt procedure does its task then unblocks driver that started it
- Steps must be performed in software upon occurance of an interrupt
  - Save registers not already saved by hardware interrupt mechanism
  - Set up context (address space) for interrupt service procedure
    - Typically, handler runs in the context of the currently running process; No expensive context switch
  - Set up stack for interrupt service procedure
    - Handler usually runs on the kernel stack of current process
    - Implies handler cannot block as the unlucky current process will also be blocked ⇒ might cause deadlock
  - Ack/Mask interrupt controller, reenable other interrupts
  - Run interrupt service procedure
    - Acknowledges interrupt at device level
    - Figures out what caused the interrupt; Received a network packet, disk read finished, UART transmit queue empty
    - If needed, it signals blocked device driver
  - In some cases, will have woken up a higher priority blocked thread
    - Choose newly woken thread to schedule next.
    - Set up MMU context for process to run next
  - Load new/original process' registers
  - Start running the new process

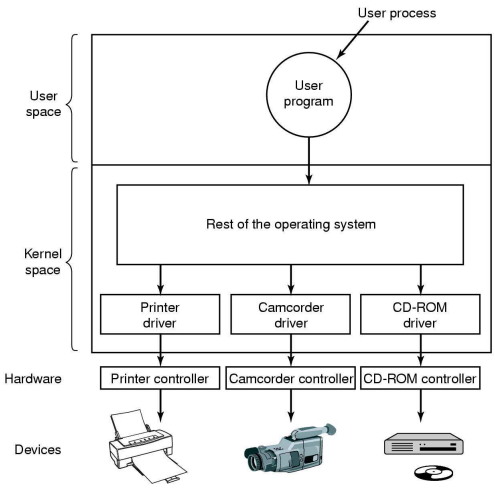


Figure 5.9: Logical positioning of device drivers. In reality all communications between drivers and device controllers goes over the bus.

5.4.2 Device Drivers (see Fig. 5.9)

- Drivers (originally) compiled into the kernel
  - Device installers were technicians
  - Number and types of devices rarely changed
- Nowadays they are dynamically loaded when needed
  - Linux modules
  - Typical users (device installers) can't build kernels
  - Number and types vary greatly; Even while OS is running (e.g hot-plug USB devices)
- Drivers classified into similar catagories; Block devices and character (stream of data) device
- OS defines a standard (internal) interface to the different classes of devices
- Device drivers job
  - translate request through the device-independent standard interface (open, close, read, write) into appropriate sequence of commands (register manipulations) for the particular hardware

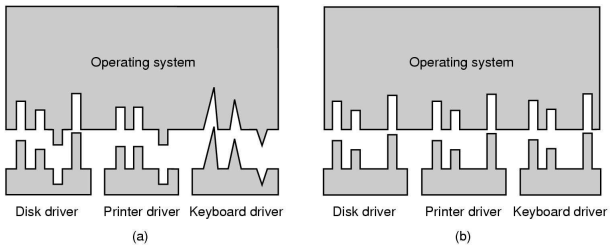


Figure 5.10: (a) Without a standard driver interface (b) With a standard driver interface.

  - Initialise the hardware at boot time, and shut it down cleanly at shutdown

- After issue the command to the device, the device either
  - Completes immediately and the driver simply return to the caller
  - Or, device must process the request and the driver usually blocks waiting for an I/O complete interrupt.
- Drivers are reentrant as they can be called by another process while a process is already blocked in the driver
  - Reentrant: Code that can be executed by more than one thread (or CPU) at the same time
  - Manages concurrency using synch primitives

5.4.3 Device Independent I/O Software(see Fig. 5.10)

- There is commonality between drivers of similar classes
- Divide I/O software into device-dependent and device-independent I/O software
- Device independent software includes
  - Buffer or Buffer-cache management
  - Managing access to dedicated devices
  - Error reporting
- Driver $\Leftrightarrow$ Kernel Interface; Major Issue is uniform interfaces to devices and kernel

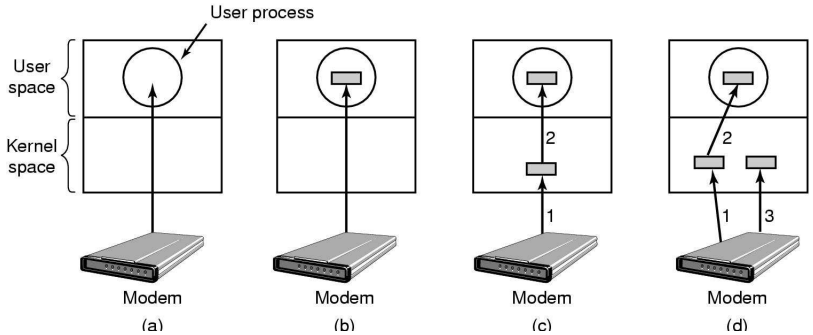


Figure 5.11: (a) Unbuffered input (b) Buffering in user space (c) *Single buffering* in the kernel followed by copying to user space (d) Double buffering in the kernel.

  - Uniform device interface for kernel code
    - Allows differents devices to be used the same way
      - No need to rewrite filesystem to switch between SCSI, IDE or RAM disk
    - Allows internal changes to device driver with fear of breaking kernel code
  - Uniform kernel interface for device code
    - Drivers use a defined interface to kernel services (e.g. kmalloc, install IRQ handler, etc.)
    - Allows kernel to evolve without breaking existing drivers
  - Together both uniform interfaces avoid a lot of programming implementing new interfaces

- No Buffering (see Fig. 5.11)
  - Process must read/write a device a byte/word at a time
  - Each individual system call adds significant overhead
  - Process must what until each I/O is complete
    - Blocking/interrupt/waking adds to overhead.
    - Many short runs of a process is inefficient (poor CPU cache temporal locality)

- User-level Buffering (see Fig. 5.11)

  - Process specifies a memory *buffer* that incoming data is placed in until it fills
    * Filling can be done by interrupt service routine
    * Only a single system call, and block/wakeup per data buffer; Much more efficient
  - Issues
    * What happens if buffer is paged out to disk
      · Could lose data while buffer is paged in
      · Could lock buffer in memory (needed for DMA), however many processes doing I/O reduce RAM available for paging. Can cause deadlock as RAM is limited resource
    * Consider write case, When is buffer available for re-use?
      · Either process must block until potential slow device drains buffer
      · or deal with asynchronous signals indicating buffer drained

- Single Buffer (see Fig. 5.11)
  - Operating system assigns a buffer in main memory for an I/O request
  - Stream-oriented
    * Used a line at time
    * User input from a terminal is one line at a time with carriage return signaling the end of the line
    * Output to the terminal is one line at a time
  - Block-oriented
    * Input transfers made to buffer
    * Block moved to user space when needed
    * Another block is moved into the buffer; Read ahead
    * User process can process one block of data while next block is read in
    * Swapping can occur since input is taking place in system memory, not user memory
    * Operating system keeps track of assignment of system buffers to user processes
  - What happens if kernel buffer is full, the user buffer is swapped out, and more data is received??? We start to lose characters or drop network packets

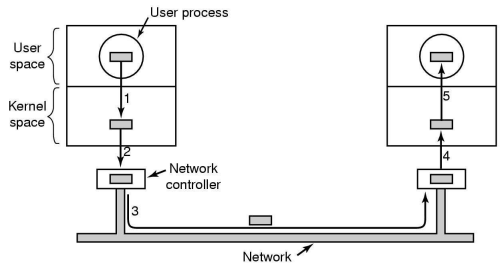


Figure 5.12: Networking may involve many copies.

- Double Buffer (see Fig. 5.11)
  - Use two system buffers instead of one
  - A process can transfer data to or from one buffer while the operating system empties or fills the other buffer
  - May be insufficient for really bursty traffic
    - Lots of application writes between long periods of computation
    - Long periods of application computation while receiving data
    - Might want to read-ahead more than a single block for disk
- Notice that buffering, double buffering are all Bounded-Buffer Producer-Consumer Problems
- Buffering in Fast Networks (see Fig. 5.12)
  - Copying reduces performance; Especially if copy costs are similar to or greater than computation or transfer costs
  - Super-fast networks put significant effort into achieving zero-copy
  - Buffering also increases latency

5.4.4 User Level Software

- library calls
  - users generally make library cals that then make the system calls
  - example:
    - int count=**write**(fd,buffer,n);
    - **write** function is run at the user level

        * simply takes parameters and makes a system call
    - another example:
        * `printf("My age: %d \n",age);`
        * takes a string, reformats it, and then calls the write system call

- spooling
    - user program places data in a special directory
    - a *daemon* (background program) takes data from directoryy and outputs it to a device
        * the user doesn't have permission to directly access the device
        * daemon runs as a privileged user
    - prevents users from tying up resources for extented periods of time; printer example
    - OS never has to get involved in working with the I/O device

5.5 Disks (see Fig. 5.13)

- Management and ordering of disk access requests is important:
    - Huge speed gap between memory and disk
    - Disk throughput is extremely sensitive to
        * Request order $\Longrightarrow$ Disk Scheduling
        * Placement of data on the disk $\Longrightarrow$ file system design
    - Disk scheduler must be aware of *disk geometry*

- Disk management issues
    - Formatting
        * Physical: divide the blank slate into sectors identified by headers containing such information as sector number; sector interleaving
        * Logical: marking bad blocks; partitioning (optional) and writing a blank directory on disk; installing file allocation tables, and other relevant information (file system initialization)
    - Reliability

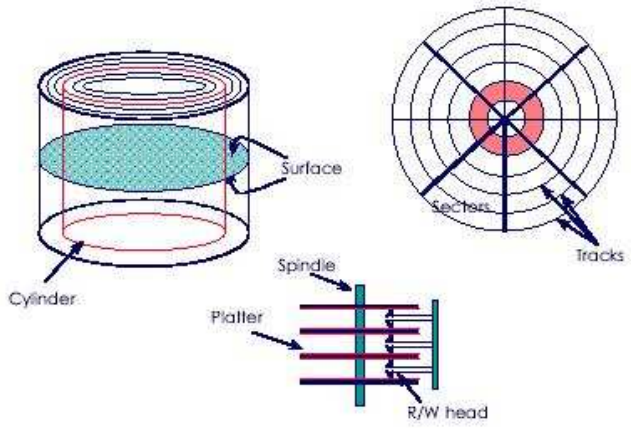
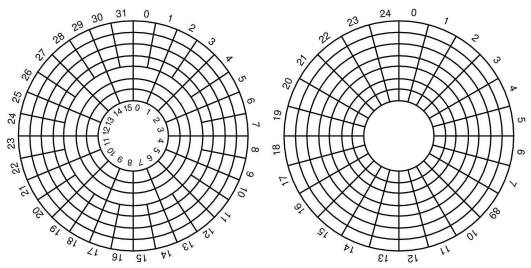


Figure 5.13: Disk Structure.

  - ∗ disk interleaving or striping
  - ∗ RAIDs (Redundant Array of Inexpensive Disks): various levels, e.g., level 0 is disk striping)
- – Controller caches newer disks have on-disk caches (128KB 512KB)

5.5.1 Disk Hardware

- Disk drives addressed as large 1-dimensional arrays of *logical blocks* (smallest transfer unit)
- 1-dimensional array of logical blocks mapped onto sectors of disk sequentially
  - – sector 0: 1st sector of 1st track on outermost cylinder
  - – mapping in order through that track, then rest of tracks in that cylinder, then through rest of cylinders from outermost to innermost
- Outer tracks can store more sectors than inner without exceed max information density (see Fig. 5.14 Left)
- Evolution of Disk Hardware (see Fig. 5.14 Right)
  - – Average seek time is approx 12 times better
  - – Rotation time is 24 times faster



Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 µsec

Figure 5.14: Left: (a) Physical geometry of a disk with two zones (b) A possible virtual geometry for this disk, Right: Disk parameters for the original IBM PC floppy disk and a Western Digital WD 18300 hard disk.

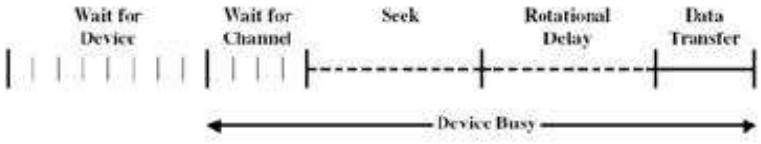


Figure 5.15: Disk Performance.

  - Transfer time is 1300 times faster
  - Most of this gain is due to increase in density
  - Represents a gradual engineering improvement

- Disk Performance (see Fig. 5.15)
  - Disk is a moving device; must be positioned correctly for I/O
  - Execution of a disk operation involves
    * Wait time: the process waits to be granted device access
      · Wait for device: time the request spend in wait queue
      · Wait for channel: time until a shared I/O channel is available
    * Access time: time hardware need to position the head
      · Seek time: position the head at the desire track
      · Rotational delay (latency): spin disk to the desired sector
    * Transfer time: sectors to be read/written rotate below head

- Estimating Access Time;

  - Seek Time T_s: Moving the head to the required tgrack not linear in the number of tracks to traverse: startup time, settling time. Typical avearge seek time: a few milliseconds
  - Rotational delay: rotational speed, r, of 5000 to 10000 rpm. At 10000 rpm, one revolution per 6ms⇒ average delay 3ms
  - Transfer time: to transfer b bytes, with N bytes per track;

$$T = \frac{b}{rN}$$

    Total average access time:

$$T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$$

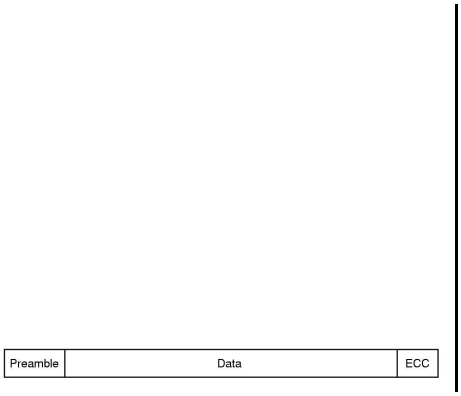
- A Timing Comparison
  - $T_s = 2$ ms, $r = 10000$rpm, 512B sect, 320 sect/track
  - read a file with 2560 sectors (=1.3MB)
  - file stored compactly (8 adjacent tracks): Read first track

Average seek	2ms
Rot. Delay	3ms
Read 320 sectors	6ms
Total	11ms
All sectors	11+7*9=74ms

  - Sectors distributed randomly over the disk: Read any sector

Average seek	2ms
Rot. Delay	3ms
Read 1 sectors	0.01875ms
Total	5.01875ms
All	2560*5.01875=20,328ms

- Disk Performance is Entirely Dominated by Seek and Rotational Delays
  - Will only get worse as capacity increases much faster than increase in seek time and rotation speed (it has been easier to spin the disk faster than improve seek time)
  - Operating System should minimise mechanical delays as much as possible



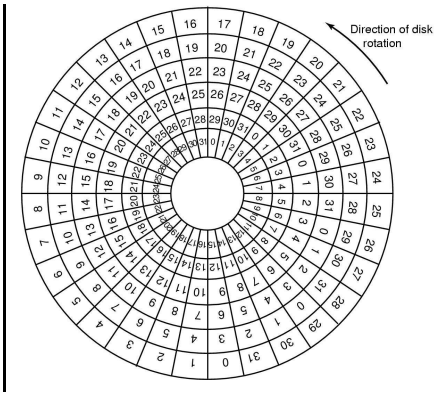


Figure 5.16: Left: Low-level Disk Formatting; A disk sector, Right: An illustration of cylinder skew.

5.5.2 Disk Formatting

- A hard disk consist of a stack of aliminum, aaloy, or glass platters 5.25 inch or 3.5 inch in diameter. On each platter is deposited a thin magnetizable metal oxide
- Before the disk can be used, each platter must recieve a **low-level format , or physical formatting** ; divide disk into sectors that disk controller can read and write (see Fig. 5.16 left)
- To use disk to hold files, OS needs to record own data structures on disk
  - *partition* disk into ≥ 1 groups of cylinders *logical formatting* or "making a file system"
- Boot block to start up system
  - bootstrap code in ROM
  - *bootstrap loader* program minimum in ROM
- When reading sequential blocks, the seek time can result in missing block 0 in the next track
- Disk can be formatted using a cylinder *skew* to avoid this (see Fig. 5.16 right)
- Issue: After reading one sector, the time it takes to transfer the data to the OS and receive the next request results in missing reading the next sector

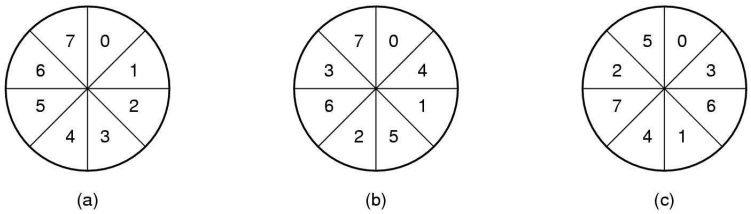


Figure 5.17: a) No interleaving b) Single interleaving c) Double interleaving.

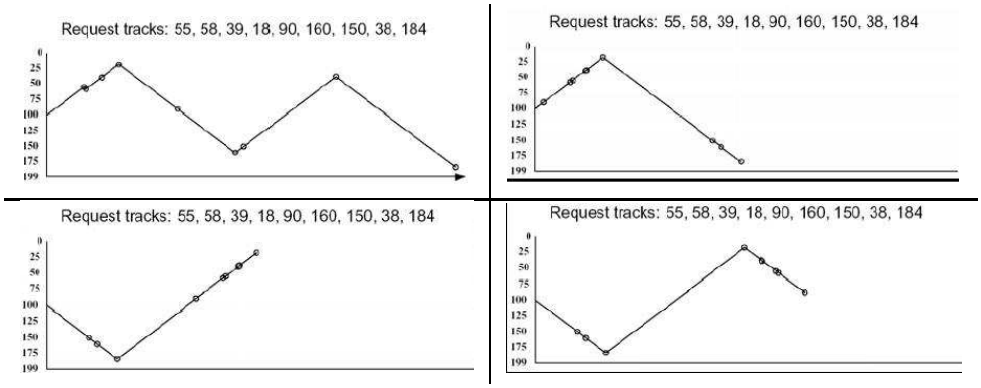


Figure 5.18: From left to right: First-in, First-out (FIFO); Shortest Seek Time First; Elevator Algorithm (SCAN); Modified Elevator (Circular SCAN, C-SCAN)

- To overcome this, we can use interleaving (see Fig. 5.17)
- Modern drives overcome interleaving type issues by simply reading the entire track (or part thereof) into the on-disk controller and caching it.

5.5.3 Disk Arm Scheduling Algorithms (see Fig. 5.18)

- Time required to read or write a disk block determined by 3 factors; Seek time, Rotational delay, Actual transfer time
- Seek time dominates
- For a single disk, there will be a number of I/O requests
- Processing them in random order leads to worst possible performance
- **First-in, First-out (FIFO)**

  - Process requests as they come
  - Fair (no starvation)
  - Good for a few processes with clustered requests
  - Deteriorates to random if there are many processes

- **Shortest Seek Time First**
  - Select request that minimises the seek time
  - Generally performs much better than FIFO
  - May lead to starvation

- **Elevator Algorithm (SCAN)**
  - Move head in one direction; Services requests in track order until it reaches the last track, then reverses direction
  - Better than FIFO, usually worse than SSTF
  - Avoids starvation
  - Makes poor use of sequential reads (on down-scan)

- **Modified Elevator (Circular SCAN, C-SCAN)**
  - Like elevator, but reads sectors in only one direction; When reaching last track, go back to first track non-stop
  - Better locality on sequential reads
  - Better use of read ahead cache on controller
  - Reduces max delay to read a particular sector

- Selecting a Disk-Scheduling Algorithm
  - SSTF common, natural appeal
  - SCAN and C-SCAN perform better if heavy load on disk
  - Performance depends on number and types of requests
  - Requests for disk service influenced by file-allocation method
  - Disk-scheduling should be separate module of OS, allowing replacement with different algorithm if necessary

5.5.4 Error–Handling

- Bad blocks are usually handled transparently by the on-disk controller (see Fig. 5.19)

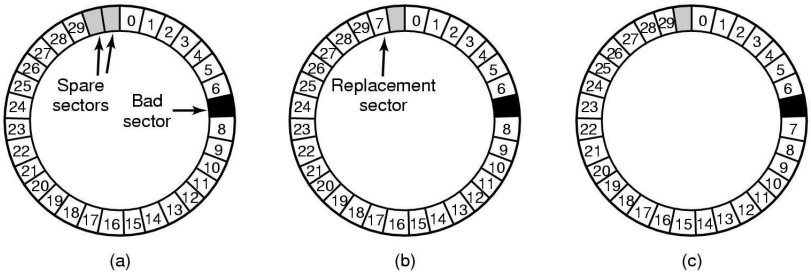


Figure 5.19: a) A disk track with a bad sector b) Substituting a spare for the bad sector c) Shifting all the sectors to bypass the bad one.

Chapter 6

File Systems

6.1 Files

- A file is a named collection of related information, usually as a sequence of bytes, with two views:
  - Logical (programmer's) view, as the users see it.
  - Physical (operating system) view, as it actually resides on secondary storage.
- What is the difference between a file and a data structure in memory? Basically,
  - files are intended to be non-volatile; hence in principle, they are long lasting,
  - files are intended to be moved around (i.e., copied from one place to another), accessed by different programs and users, and so on.
- File lifetime is independent of process lifetime
- Used to share data between processes
- Input to applications is by means of a file
- **File Management**; File management system is considered part of the operating system
  - Manages a trusted, shared resource
  - Bridges the gap between:
    - lowlevel disk organization (an array of blocks),

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

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 compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Figure 6.1: Some typical file extensions.

    - ∗ and the user's views (a stream or collection of records)
  - Also includes tools outside the kernel; E.g. formatting, recovery, defrag, consistency, and backup utilities.
  - Objectives for a File Management System;
    - ∗ Provide a convenient naming system for files
    - ∗ Provide uniform I/O support for a variety of storage device types
    - ∗ Provide a standardized set of I/O interface routines
    - ∗ Guarantee that the data in the file are valid
    - ∗ Optimize performance
    - ∗ Minimize or eliminate the potential for lost or destroyed data
    - ∗ Provide I/O support and access control for multiple users
    - ∗ Support system administration (e.g., backups)

6.1.1 File Naming

- File system must provide a convenient naming scheme
- Textual Names (see Fig. 6.1)
- May have restrictions
  - Only certain characters, E.g. no '/' characters
  - Limited length

  - Only certain format, E.g DOS, 8 + 3
- Case (in)sensitive
- Names may obey conventions (.c files or C files)
  - Interpreted by tools (UNIX)
  - Interpreted by operating system (Windows)

6.1.2 File Structure; From OS's perspective

- **Stream of Bytes** (see Fig. 6.2)
  - OS considers a file to be unstructured
  - Simplifies file management for the OS
  - Applications can impose their own structure
  - Used by UNIX, Windows, most modern OSes
- **Records** (see Fig. 6.2)
  - Collection of bytes treated as a unit; Example: employee record
  - Operations at the level of records (read_rec, write_rec)
  - File is a collection of similar records
  - OS can optimize operations on records
- **Tree of Records** (see Fig. 6.2)
  - Records of variable length
  - Each has an associated key
  - Record retrieval based on key

6.1.3 File Types

- Regular files
- Directories
- Device Files
  - Character Devices

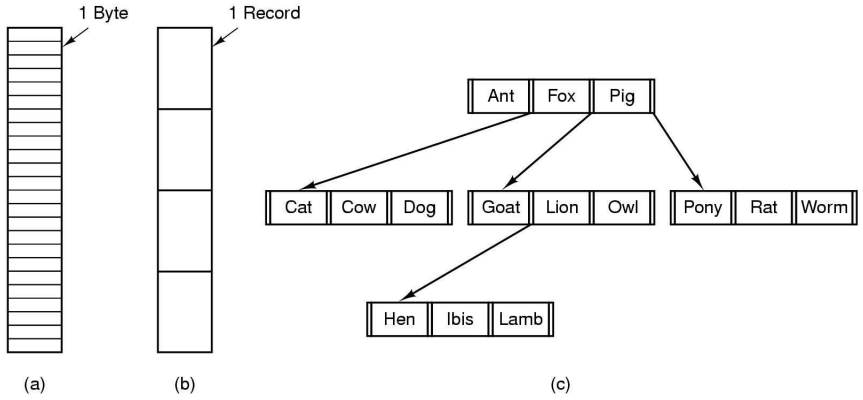


Figure 6.2: Three kinds of files. (a) byte sequence. (b) record sequence. (c) tree.

  - Block Devices

- Some systems distinguish between regular file types; ASCII text files, binary files
- A common implementation technique (as organizational help with consistent usage) is to include the type as an extension to the file name (see Fig. 6.1)
- Files are structured internally to meet the expectations of the program(s) that manipulate them.
- All systems recognize their own executable file format; May use a magic number (see Fig. 6.3)

6.1.4 File Access

- The information stored in a file can be accessed in a variety of methods:
  - Sequential access
    * in order, one record after another
    * read all bytes/records from the beginning
    * cannot jump around, could rewind or back up
    * convenient when medium was mag tape
  - Random (Direct) access

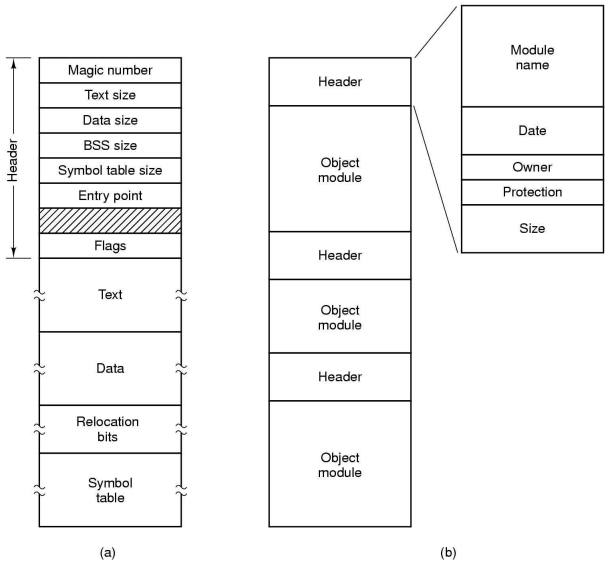


Figure 6.3: (a) An executable file (b) An archive.

* bytes/records read in any order skipping the previous records
* essential for data base systems
* read can be
  - move file pointer (seek), then read or
  - each read specifies the file pointer

– Keyed; in any order, but with particular value(s); e.g., hash table or dictionary. TLB lookup is one example of a keyed search

- Other access methods, such as indexed, can be built on top of the above basic techniques.

6.1.5 File Attributes(see Fig. 6.4)

- Information about files kept in directory structure maintained on disk
- Each file is associated with a collection of information, known as attributes:
  – name, owner, creator, only information in human-readable form
  – type, (e.g., source, data, binary) needed if system supports different types

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 6.4: Some possible file attributes.

- location, (e.g., I-node or disk address) pointer to file location on device
- organization, (e.g., sequential, indexed, random)
- time and date, (creation, modification, and last accessed)
- size, current file size
- protection, who can do reading, writing, executing
- variety of other (e.g., maintenance) information.

6.1.6 File Operations

- There are six basic operations (not all) for file manipulation: create, write, read, delete, reposition r/w pointer (a.k.a. seek), and truncate (not very common.) (see Fig. 6.5)
- Open(Fi) search directory structure on disk for entry Fi, move content of entry to memory
- Close (Fi) move content of entry Fi in memory to directory structure on disk

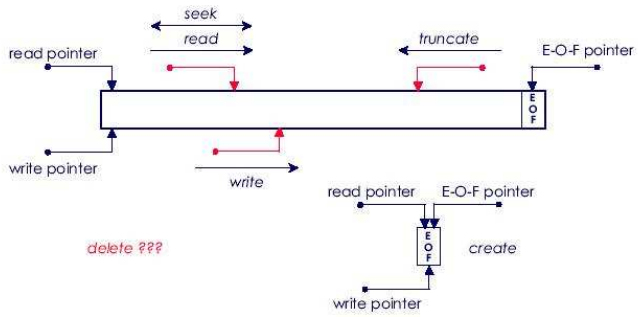


Figure 6.5: File operations.

6.1.7 An Example Program Using File System Calls (see Fig. 6.6)

- copyfile **abc** *xyz*; where argv[0]="copyfile", argv[1]="abc", argv[2]="xyz"

6.1.8 Memory–Mapped Files (see Fig. 6.7)

- Avoids translating from on-disk format to in-memory format (and vice versa)
  - Supports complex structures
  - No read/write systems calls
  - File simply (paged or swapped) to file system
  - Unmap when finished
- Problems
  - Determining actual file size after modification; Round to nearest whole page (even if only 1 byte file)
  - Care must be taken if file is shared; E.g. one process memorymapped and one process read/write syscalls
  - Large files may not fit in the virtual address space

6.1.9 File Organization and Access; Programmer's Perspective

- One of the key elements of a file system is the way the files are organized. File organization is the logical structuring as well as the access method(s) of files.

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

Figure 6.6: A simple program to copy a file.

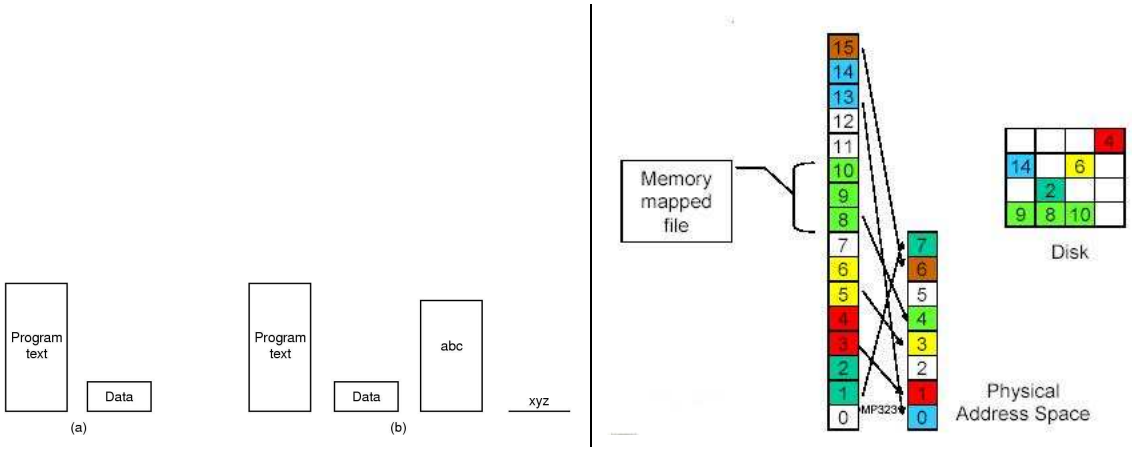


Figure 6.7: Left: (a) Segmented process before mapping files into its address space (b) Process after mapping existing file abc into one segment creating new segment for xyz. Right: Memory mapped files and paging

- Given an operating system supporting unstructured files that are stream-of-bytes, how should one organize the contents of the files?
- Performance considerations:
  - File system performance affects overall system performance
  - Organization of the file system affects performance
  - File organization (data layout) affects performance; depends on access patterns
- Possible access patterns:
  - Read the whole file
  - Read individual blocks or records from a file
  - Read blocks or records preceding or following the current one
  - Retrieve a set of records
  - Write a whole file sequentially
  - Insert/delete/update records in a file
  - Update blocks in a file
- Criteria for File Organization
  - Rapid access

    - Needed when accessing a single record
    - Not needed for batch mode
  - Ease of update; File on CDROM will not be updated, so this is not a concern
  - Economy of storage
    - Should be minimum redundancy in the data
    - Redundancy can be used to speed access such as an index
  - Simple maintenance
  - Reliability

- Fundamental File Organizations; Common file organization schemes are:
  - Pile
  - Sequential
  - Indexed Sequential
  - Indexed
  - Direct or Hashed

- **Pile** (see Fig. 6.8)
  - Data are collected in the order they arrive
  - Purpose is to accumulate a mass of data and save it
  - Records may have different fields
  - No structure
  - Record access is by exhaustive search
  - Update:
    - Same size record; okay
    - Variable size; poor
  - Retrieval:
    - Single record; poor
    - Subset; poor
    - Exhaustive; okay

- **Sequential** (see Fig. 6.8)
  - Fixed format used for records

- Records are the same length
- Field names and lengths are attributes of the file
- One field is the key filed
  - Uniquely identifies the record
  - Records are stored in key sequence
- New records are placed in a log file or transaction file
- Batch update is performed to merge the log file with the master file
- Update:
  - Same size record; good
  - Variable size; No
- Retrieval:
  - Single record; poor
  - Subset; poor
  - Exhaustive; okay

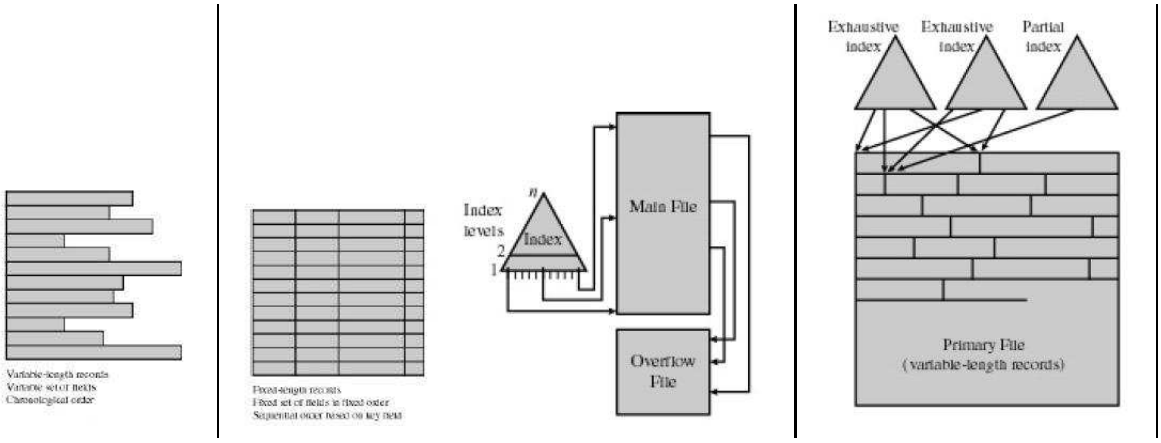


Figure 6.8: Fundamental File Organizations; (a) Pile (b) Sequential (c) Indexed Sequential (d) Indexed.

- **Indexed Sequential** (see Figs. 6.8,6.9)
  - Index provides a lookup capability to quickly reach the vicinity of the desired record
    - Contains key field and a pointer to the main file

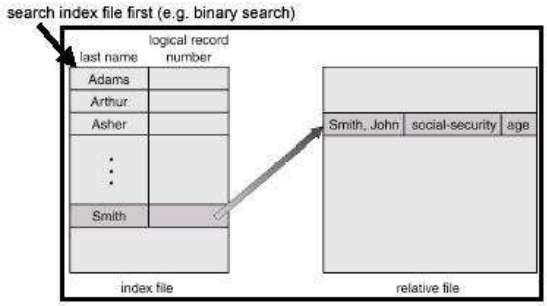


Figure 6.9: IBM indexed-sequential access method (ISAM).

    * Indexed is searched to find highest key value that is equal or less than the desired key value
    * Search continues in the main file at the location indicated by the pointer
  - New records are added to an overflow file
  - Record in main file that precedes it is updated to contain a pointer to the new record
  - The overflow is merged with the main file during a batch update
  - Update:
    * Same size record; good
    * Variable size; No
  - Retrieval:
    * Single record; good
    * Subset; poor
    * Exhaustive; okay

- **Comparison of sequential and indexed sequential lookup**
  - Example: a file contains 1 million records
  - On average 500,00 accesses are required to find a record in a sequential file
  - If an index contains 1000 entries, it will take on average 500 accesses to find the key, followed by 500 accesses in the main file. Now on average it is 1000 accesses

- **Indexed File** (see Fig. 6.8)

  - Uses multiple indexes for different key fields
  - May contain an exhaustive index that contains one entry for every record in the main file
  - May contain a partial index
  - Update:
    * Same size record; good
    * Variable size; good
  - Retrieval:
    * Single record; good
    * Subset; good (Assuming the selecting attribute is indexed on)
    * Exhaustive; okay

- **The Direct, or Hashed File**
  - Key field required for each record
  - Key maps directly or via a hash mechanism to an address within the file
  - Directly access a block at a the known address
  - Update:
    * Same size record; good
    * Variable size; No (Fixed sized records used)
  - Retrieval:
    * Single record; excellent
    * Subset; poor
    * Exhaustive; poor

6.2 Directories

- A directory is a symbol table, which can be searched for information about the files. Also, it is the fundamental way of organizing files. Usually, a directory is itself a file
- A typical directory entry contains information (attributes, location, ownership) about a file. Directory entries are added as files are created, and are removed when files are deleted.
- Provides mapping between file names and the files themselves

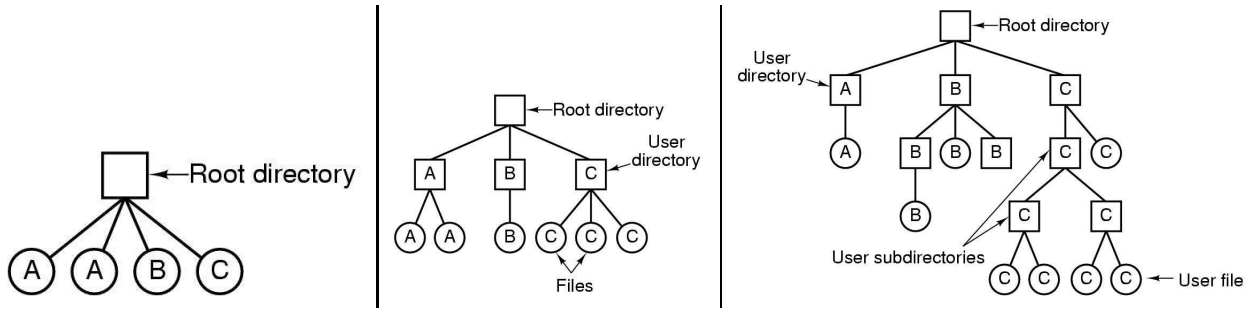


Figure 6.10: (a) Single-Level Directory Systems (b) Two-Level Directory Systems (c) Hierarchical Directory Systems.

- Goals in Organization of Directory
  - **Efficiency**; locate file quickly
  - **Naming**; convenient to users,
    - 2 users can use same name for different files
    - Same file can have several different names
  - **Grouping**; logical grouping of files by attributes, (e.g., all Java programs, all games, . . .)

- Single–Level Directory Systems (see Figs. 6.10,6.11)
  - List of entries, one for each file
  - Sequential file with the name of the file serving as the key
  - Provides no help in organizing the files
  - Forces user to be careful not to use the same name for two different files

- Two–Level Directory Systems (see Figs. 6.10,6.11)
  - One directory for each user and a master directory
  - Master directory contains entry for each user; Provides access control information
  - Each user directory is a simple list of files for that user
  - Still provides no help in structuring collections of files

- Hierarchical, or Tree-Structured Directory Systems (see Figs. 6.10,6.11)

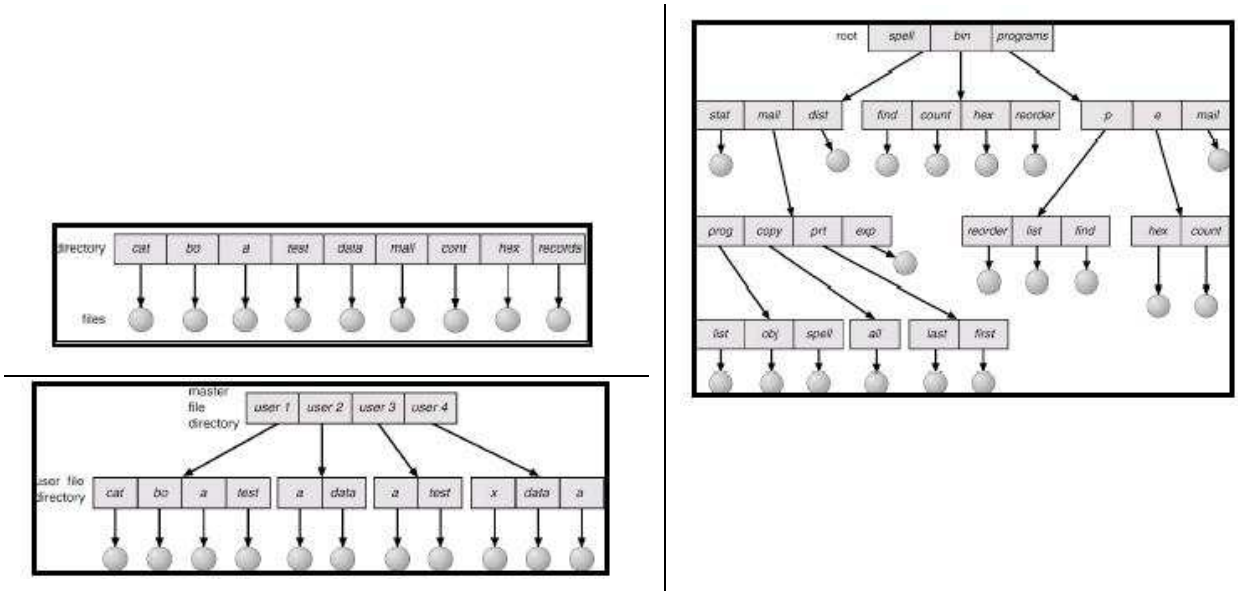


Figure 6.11: Example to (a) Single-Level Directory Systems (b) Two-Level Directory Systems (c) Hierarchical Directory Systems.

- Files can be located by following a path from the root, or master, directory down various branches; This is the absolute pathname for the file
- Can have several files with the same file name as long as they have unique path names

6.2.1 Path Names

- Always specifying the absolute pathname for a file is tedious!
- Introduce the idea of a working directory; Files are referenced relative to the working directory
- Example: cwd = /home/dizin, profile = /home/dizin/.profile
- Absolute pathname; A path specified from the root of the file system to the file
- A Relative pathname; A pathname specified from the cwd
- Note: '.' (dot) and '..' (dotdot) refer to current and parent directory
  - Example: cwd = /home/dizin

  - ../../etc/passwd
  - /etc/passwd
  - ../dizin/../../etc/passwd
  - Are all the same file

6.2.2 File Sharing

- In multiuser system, allow files to be shared among users
- Two issues
  - Access rights. Allowing users to share files raises a major issue: protection. A general approach is to provide controlled access to files through a set of operations such as read, write, delete, list, and append. Then permit users to perform one or more operations. One popular protection mechanism is a condensed version of access list, where the system recognizes three classifications of users with each file and directory: user, group, other
  - Management of simultaneous access
- **Access Rights**
  - None
    - User may not know of the existence of the file
    - User is not allowed to read the user directory that includes the file
  - Knowledge; User can only determine that the file exists and who its owner is
  - Execution; The user can load and execute a program but cannot copy it
  - Reading; The user can read the file for any purpose, including copying and execution
  - Appending; The user can add data to the file but cannot modify or delete any of the file's contents
  - Updating; The user can modify, deleted, and add to the file's data. This includes creating the file, rewriting it, and removing all or part of the data
  - Changing protection; User can change access rights granted to other users

  - Deletion; User can delete the file
  - Owners
    * Has all rights previously listed
    * May grant rights to others using the following classes of users; Specific user, User groups, All for public files

```
total 1704
drwxr-x--- 3 user group 4096 Oct 14 08:13 .
drwxr-x--- 3 user group 4096 Oct 14 08:14 ..
drwxr-x--- 2 user group 4096 Oct 14 08:12 backup
-rw-r----- 1 user group 141133 Oct 14 08:13 eniac3.jpg
-rw-r----- 1 user group 1580544 Oct 14 08:13 wk11.ppt
```

  - First letter: file type
    * **d** for directories
    * **-** for regular files
  - Three user categories; user, group, and other
  - Three access rights per category; read, write, and execute
  - d**rwx***rwx*rwx; **user** *group* other
  - Execute permission for directory? Permission to access files in the directory
  - To list a directory requires read permissions
  - What about drwxr-x–x ?
  - Problematic example
    * A owns file foo.bar
    * A wishes to keep his file private
    * Inaccessible to the general public
    * A wishes to give B read and write access
    * A wishes to give C readonly access
    * ???????

- **Management of Simultaneous Access**
  - Most Oses provide mechanisms for users to manage concurrent access to files; Example: lockf(), flock() system calls
  - Typically
    * User may lock entire file when it is to be updated
    * User may lock the individual records during the update
  - Mutual exclusion and deadlock are issues for shared access

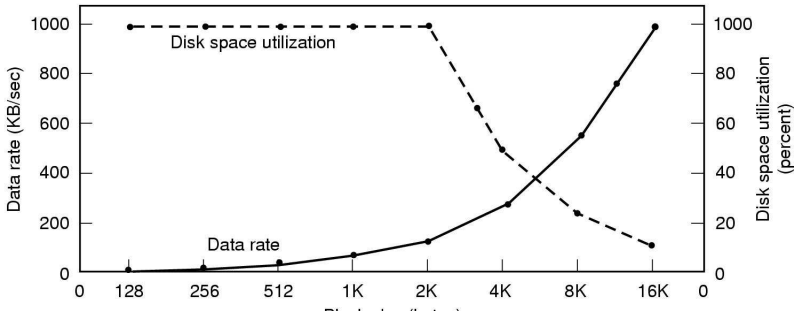


Figure 6.12: The solid curve (left hand scale) gives data rate of a disk. The dashed curve (right hand scale) gives disk space efficiency. All files 2KB (an approximate median file size).

6.3 File System Implementation

6.3.1 File System Layout

- Tradeoff in physical block size (see Fig. 6.12)
  - Sequential Access; The larger the block size, the fewer I/O operation required
  - Random Access
    - The larger the block size, the more unrelated data loaded.
    - Spatial locality of access improves the situation.
  - Choosing the an appropriate block size is a compromise

6.3.2 Implementing Files (see Fig. 6.13)

- The file system must keep track of
  - which blocks belong to which files.
  - in what order the blocks form the file
  - which blocks are free for allocation
- Given a logical region of a file, the file system must identify the corresponding block(s) on disk.
  - Stored in file allocation table (FAT) (see Fig. 6.13), directory, Inode

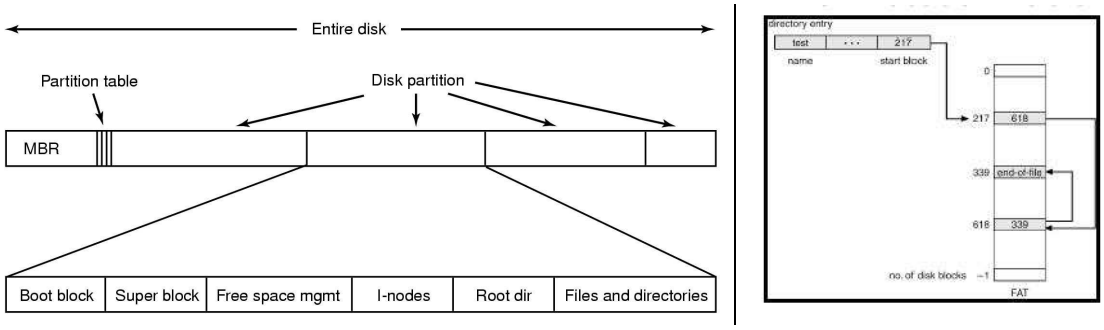


Figure 6.13: Left: A possible file system layout. Right: File Allocation Table.

- Creating and writing files allocates blocks on disk
- **Allocation Strategies**
  - Preallocation
    - Need the maximum size for the file at the time of creation
    - Difficult to reliably estimate the maximum potential size of the file
    - Tend to overestimated file size so as not to run out of space
  - Dynamic Allocation; Allocated in portions as needed
- **Portion Size**
  - Extremes
    - Portion size = length of file (remember or contiguous)
    - Portion size = block size
  - Tradeoffs
    - Contiguity increases performance for sequential operations
    - Many small portion increase the size of the file allocation table
    - Fixed–sized portions simplify reallocation of space
    - Variable–sized portions minimize internal fragmentation losses
- **Methods of File Allocation**
  - The file system allocates disk space, when a file is created. With many files residing on the same disk, the main problem is how to allocate space for them. File allocation scheme has impact on the efficient use of disk space and file access time.
  - Common file allocation techniques are:

    - Contiguous
    - Chained (linked)
    - Indexed
  - All these techniques allocate disk space on a per block (smallest addressable disk unit) basis.

- **Methods of File Allocation; Contiguous allocation** (see Fig. 6.14a, b)
  - Allocate disk space like paged, segmented memory. Keep a free list of unused disk space.
  - Single set of blocks is allocated to a file at the time of creation
  - Advantages;
    - Only a single entry in the directory entry; Starting block and length of the file
    - easy access, both sequential and random
    - Simple, only starting location (block #) and length (number of blocks) to find all contents
    - few seeks
  - Disadvantages;
    - External fragmentation will occur
    - May not know the file size in advance
    - Eventually, we will need compaction to reclaim unusable disk space.
    - Files can't grow

- **Methods of File Allocation; Chained (or linked list) allocation** (see Fig. 6.14c,d,e,f)
  - Space allocation is similar to page frame allocation. Mark allocated blocks as in-use
  - Each block contains a pointer to the next block in the chain
  - Only single entry in a directory entry; Starting block and length of file
  - No external fragmentation; Free-space manageable, no wasted space
  - Files can grow easily

  - Simple; need only starting address
  - Best for sequential files; Poor for random access
  - No accommodation of the principle of locality; Blocks end up scattered across the disk
  - To improve performance, we can run a defragmentation utility to consolidate files.
  - Storing a file as a linked list of disk blocks (see Fig. 6.14e)
  - Linked allocation with file allocation table (FAT) in RAM (see Fig. 6.14f)
    * Avoids disk accesses when searching for a block
    * Entire block is available for data
    * Table gets too large for large file systems; Can cache parts of it, but still can consume significant RAM or generate disk traffic
    * Used in MSDOS, OS/2

- **Methods of File Allocation; Indexed allocation** (see Fig. 6.15)
  - Allocate an array of pointers during file creation. Fill the array as new disk blocks are assigned
  - File allocation table contains a separate one level index for each file
  - The index has one entry for each portion allocated to the file
  - The file allocation table contains block number for the index
  - Supports both sequential and direct (easy) access to the file
  - Small internal fragmentation
  - Lots of seeks if the file is big
  - Maximum file size is limited to the size of a block
  - Portions
    * Block sized; Eliminates external fragmentation
    * Variable sized; Improves contiguity, Reduces index size
  - Most popular of the three forms of file allocation
  - Example: UNIX file system

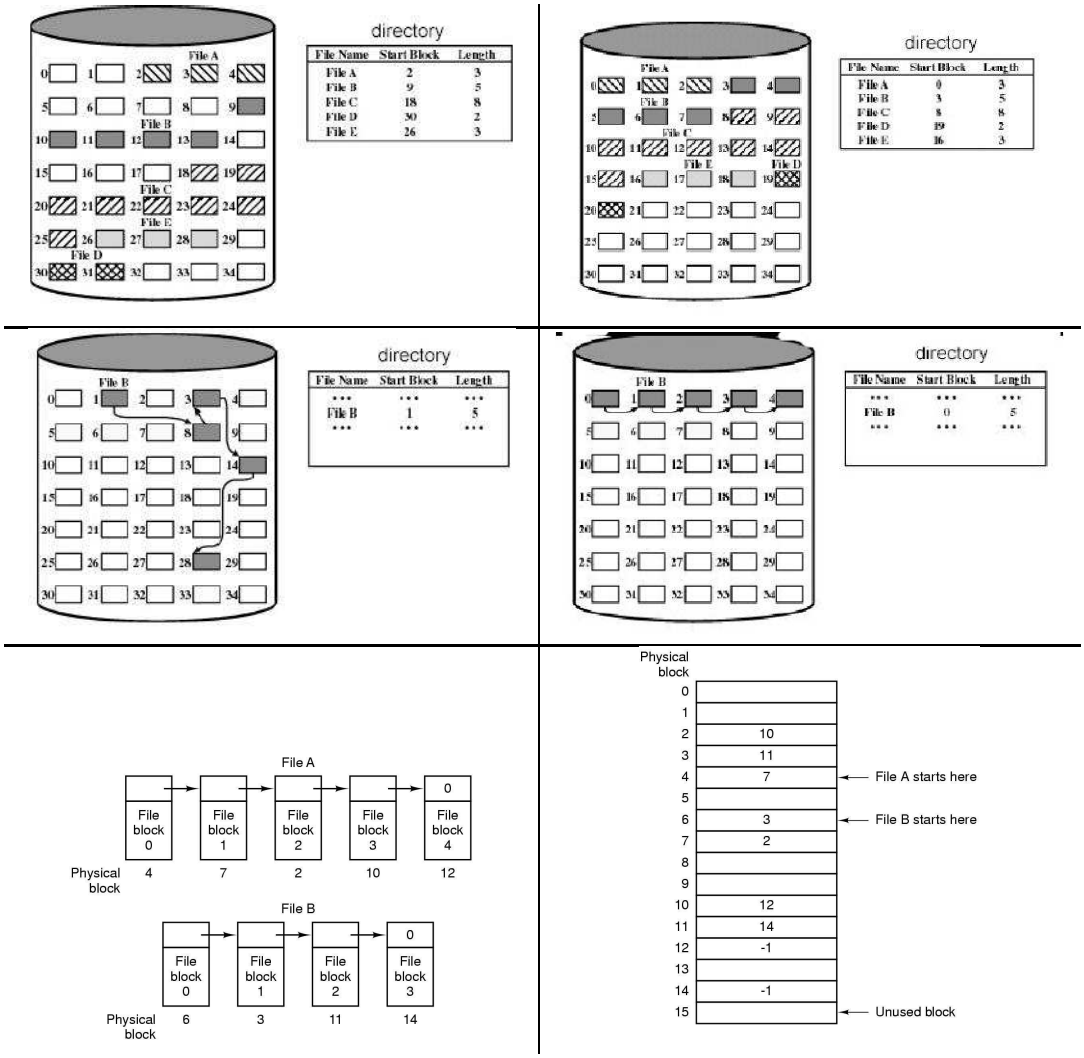


Figure 6.14: (a) Contiguous allocation (b) Contiguous allocation with compaction (c) Storing a file as a linked list of disk blocks (d) Storing a file as a linked list of disk blocks with defragmentation (e) Alternative representation of chained allocation (f) Linked list allocation using a file allocation table in main memory.

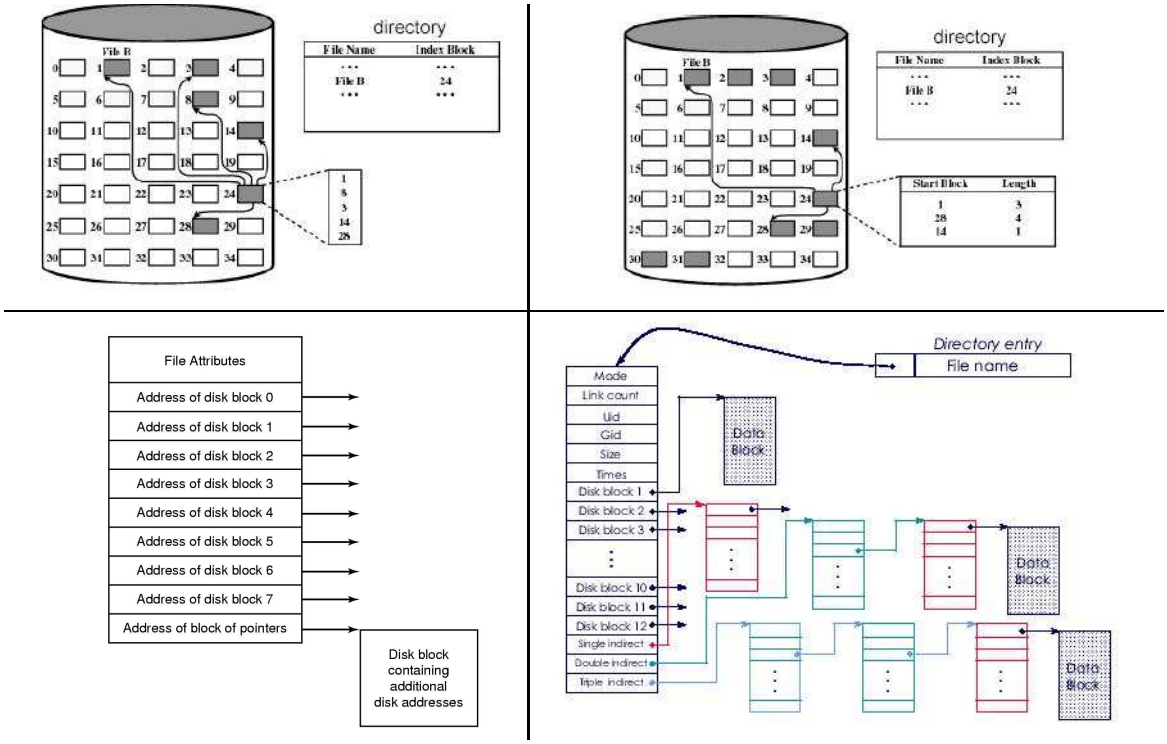


Figure 6.15: (a) Indexed allocation with block partitions (b) Indexed allocation with variable–length partitions (c) An example of i-node.

6.3.3 Implementing Directories

- (see Fig. 6.16 Left)
- A simple directory containing fixed–sized entries with the disk addresses and attributes in directory entry; **DOS/Windows**
- A directory in which each entry just refers to an inode; **UNIX**
- Fixed Size Directory Entries;
  - Either too small; Example: DOS 8+3 characters
  - Waste too much space; Example: 255 characters per file name
- Free variable length entries can create external fragmentation in directory blocks; Can compact when block is in RAM

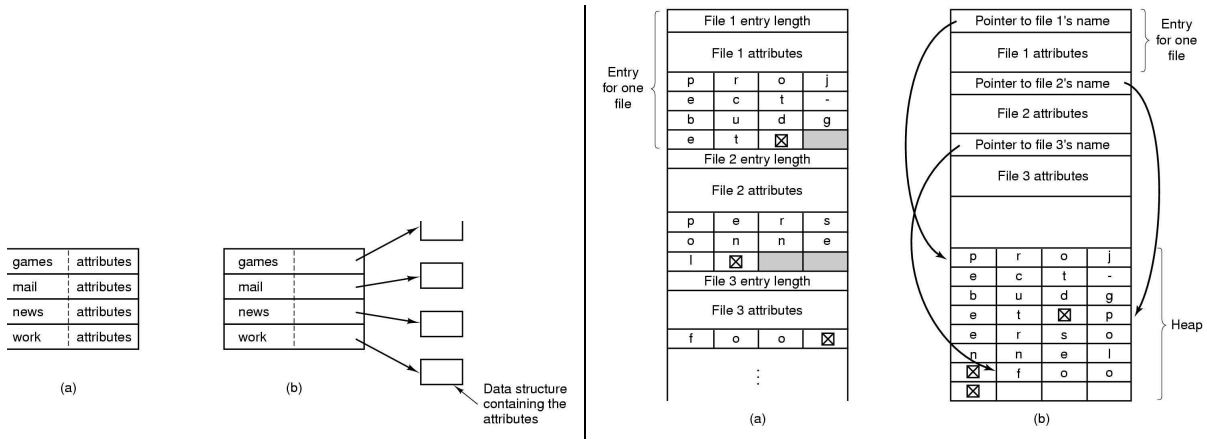


Figure 6.16: Left: (a) A simple directory containing fixed–sized entries with the disk addresses and attributes in directory entry (b) A directory in which each entry just refers to an inode. Right: Two ways of handling long file names in directory (a) Inline (b) In a heap.

6.3.4 Shared Files (see Fig. 6.17)

- Copy entire directory entry (including file attributes)
  - Updates to shared file not seen by all parties
  - Not useful
- Keep attributes separate (in Inode) and create a new entry that points to the attributes (hard link)
  - Updates visible
  - If one link remove, the other remains (ownership is an issue)
- Create a special "LINK" file that contains the pathname of the shared file (symbolic link)
  - File removal leaves dangling links
  - Not as efficient to access

6.3.5 Disk Space Management (Free space management)

- Since the amount of disk space is limited (posing a management problem similar to that of physical memory), it is necessary to reuse the space released by deleted files

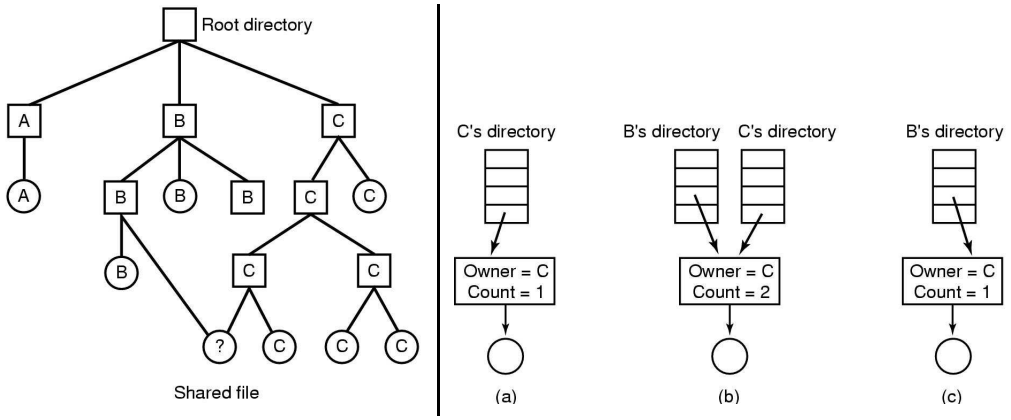


Figure 6.17: Left: File system containing a shared file. Right: (a) Situation prior to linking (b) After the link is created (c)After the original owner removes the file.

- In general, file systems keep a list of free disk blocks (initially, all the blocks are free) and manage this list by one of the following techniques
- **Bit tables** (see Fig. 6.18b)
  - Individual bits in a bit vector flags used/free blocks
  - 16GB disk with 1KB blocks; for each block 1 bit is used $(16Gb/1KB) \approx 2^{24}/8/1KB = 2048$ blocks; 2MB table
  - 16GB disk with 512byte blocks 4MB table
  - May be too large to hold in main memory
  - Expensive to search; But may use a two level table
- **Chained free portions** (see Fig. 6.18a)
  - Free portions are linked
  - Fragmentation if using variable allocation ⇒ many small portions
  - Required read before write to a block
- **Free block list**
  - Single list of a set of free block lists (unallocated blocks)
  - Manage as LIFO or FIFO on disk with ends in main memory
  - Background jobs can reorder list for better contiguity

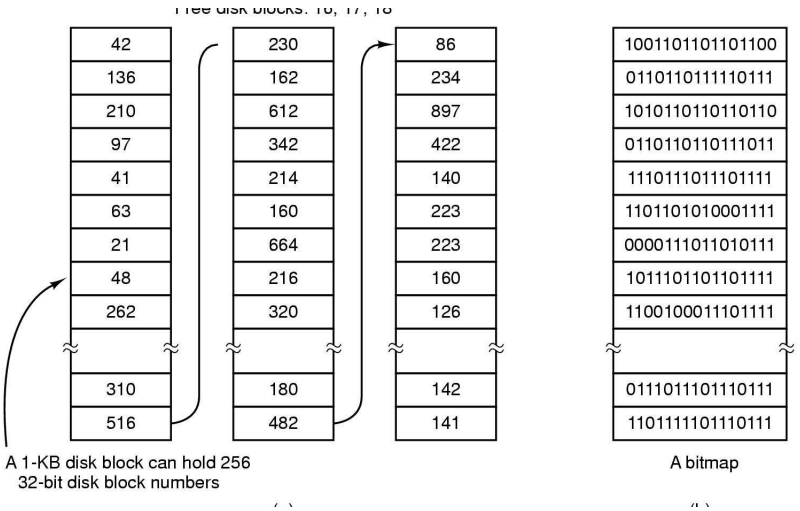


Figure 6.18: (a) Storing the free list on a linked list (b) A bit map.

6.3.6 Other file system implementation issues

- Recovery, Reliability
  - Consistency checking; compares data in directory with data blocks on disk, tries to fix inconsistencies (e.g., UNIX **fsck**)
  - Use system programs to back up data from disk to another device (optical media, magnetic tape) (disaster scenarios)
  - Recover lost file or disk by restoring from backup
- Efficiency and Performance
  - Efficiency dependent on:
    - disk allocation and directory algorithms
    - types of data kept in file's directory entry
  - Performance: disk and page caches
    - disk cache: frequently used blocks in main memory
    - free-behind, read-ahead: optimize sequential access
    - improve PC performance by dedicating section of memory as virtual disk, or RAM disk.
- Log Structured File Systems

  - Log structured (or journaling) file systems record each update to file system as transaction.
  - All transactions written to log. Transaction considered committed once written to log file system may not yet be updated.
  - Transactions in log asynchronously written to file system. When file system modified, transaction removed from log.
  - If file system crashes, all remaining logged transactions must still be performed

- Sun Network File System (NFS)
  - Implementation and specification of software system for accessing remote files across LANs (or WANs)
  - Implemented as part of Solaris, SunOS on Suns: uses unreliable datagram protocol (UDP/IP and Ethernet)
  - Widely used in UNIX (including free kinds)
  - Interconnected machines seen as independent with independent file systems allowing transparent sharing
    * remote directory mounted over local file system directory; mounted directory looks like integral subtree of local file system, replacing subtree descending from local directory
    * specification of remote directory for mount operation non-transparent; host name of remote directory provided: files in remote directory can then be accessed transparently
    * subject to access-rights, potentially any file system (or directory within file system), can be mounted remotely on top of any local directory
  - NFS designed to operate in heterogeneous environment: different machines, OSes, network architectures; NFS specifications independent of these details

6.4 UNIX File Management

- Focus on two types of files
  - Ordinary files (stream of bytes)
  - Directories
- And mostly ignore the others
  - Character devices
  - Block devices
  - Named pipes
  - Sockets
  - Symbolic links
- **UNIX index node (inode)**
  - Each file is represented by an Inode
  - Inode contains all of a file's metadata
    - Access rights, owner,accounting info
    - (partial) block index table of a file
  - Each inode has a unique number (within a partition)
    - System oriented name
    - Try 'ls –i' on Unix (Linux)
  - Directories map file names to inode numbers
    - Map human–oriented to system–oriented names
    - Mapping can be many–to–one; Hard links

```
ozdogan@ozdogan:~/week12$ man ls
.
-i, --inode   print index number of each file
.
ozdogan@ozdogan:~/week12$ ls -i
toplam 128
 901649 drwxr-xr-x 3 ozdogan  ozdogan  4096 2004-05-25 15:00 ./
 885067 drwxr--r-- 5 ozdogan  ozdogan  8192 2004-05-25 14:47 ../
 901651 drwxr-xr-x 2 ozdogan  ozdogan  4096 2004-05-25 14:47 figures/
 901656 -rw-r--r-- 1 ozdogan  ozdogan  1264 2004-05-25 14:59 week12.aux
```

```
 901658 -rw-r--r-- 1 ozdogan  ozdogan  5264 2004-05-25 14:59 week12.dvi
 901655 -rw-r--r-- 1 ozdogan  ozdogan  8654 2004-05-25 14:59 week12.log
 901657 -rw-r--r-- 1 ozdogan  ozdogan    57 2004-05-25 14:59 week12.out
 901659 -rw-r--r-- 1 ozdogan  ozdogan 55968 2004-05-25 15:00 week12.ps
 901652 -rw-r--r-- 1 ozdogan  ozdogan  2153 2004-05-25 14:59 week12.tex
 901654 -rw-r--r-- 1 ozdogan  ozdogan  1939 2004-05-25 14:58 week12.tex~
 901653 -rw-r--r-- 1 ozdogan  ozdogan 13767 2004-05-25 14:47 week12.tex.backup
```

A code example for for printing out structure members of files; Try 'structuremembers *' on Unix (Linux)

```
/* structuremembers.c
print structure members of files
st_mode the type and mode of the file
st_ino
st_dev
st_rdev
st_nlink
st_uid
st_gid
st_size
st_atime
st_mtime
st_ctime
*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
main(argc,argv)
int argc; char *argv[];
{
struct stat status;
int i;
for(i=1; i < argc; i++)
if(stat (argv[i],&status))
fprintf(stderr,"Cannot stat %s \n",argv[i]);
else
printf("%15s %4.4o\n",argv[i],status.st_mode & 07777);
//printf("%15s %14d\n",argv[i],status.st_ino);
}
```

Internal sructure of week12 inode

```
Inode: 901649   Type: directory    Mode:  0755   Flags: 0x0   Generation
User:  1000   Group:  1000   Size: 4096
File ACL: 0    Directory ACL: 0
Links: 3   Blockcount: 8
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x40b33fde -- Tue May 25 15:45:18 2004
atime: 0x40b34ba7 -- Tue May 25 16:35:35 2004
mtime: 0x40b33fde -- Tue May 25 15:45:18 2004
BLOCKS:
(0):1828886
TOTAL: 1
```

Internal sructure of week12.ps inode

```
Inode: 901659   Type: regular    Mode:  0644   Flags: 0x0   Generation:
User:  1000   Group:  1000   Size: 83309
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 176
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x40b34007 -- Tue May 25 15:45:59 2004
atime: 0x40b34016 -- Tue May 25 15:46:14 2004
mtime: 0x40b34007 -- Tue May 25 15:45:59 2004
BLOCKS:
(0-11):7742-7753, (IND):7754, (12-20):7755-7763
TOTAL: 22
```

- Inode Contents (see Fig. 6.19)
  - Mode
    - Type; Regular file or directory
    - Access mode; rwxrwxrwx
  - Uid; User ID
  - Gid; Group ID
  - atime; Time of last access
  - ctime; Time when file was reference count created
  - mtime; Time when file was last modified
  - Size; Size of the file in bytes

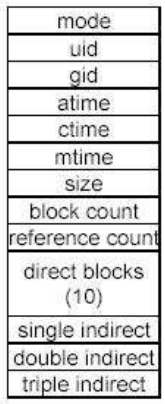


Figure 6.19: Inode contents.

- Block count; Number of disk blocks used by the file.
- Note that number of blocks can be much less than expected given the file size; Files can be sparsely populated
- Direct Blocks
  - Block numbers of first 10 blocks in the file
  - Most files are small; We can find blocks of file directly from the inode (see Fig. 6.20left)
- Problem; How do we store files greater than 10 blocks in size? Adding significantly more direct entries in the inode results in many unused entries most of the time.
- *Single Indirect Block*; Block number of a block containing block numbers, (see Fig. 6.20right) In this case 8
  - Requires two disk access to read; One for the indirect block; one for the target block
  - Max File Size
    - In previous example; 10 direct + 8 indirect = 18 block file
    - A more realistic example; Assume 1Kbyte block size, 4 byte block numbers 10 * 1K + 1K/4 * 1K = 266 Kbytes
  - For large majority of files (¡ 266 K), only one or two accesses required to read any block in file.
- Double Indirect Block; Block number of a block containing block numbers of blocks containing block numbers

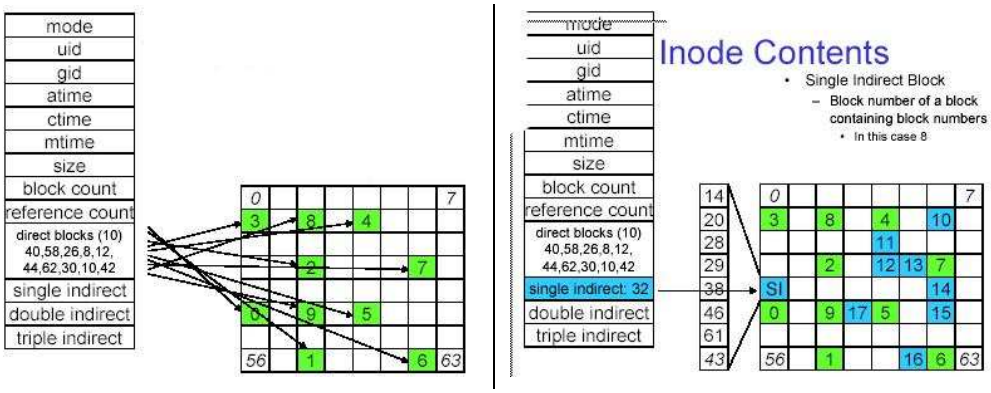


Figure 6.20: Left: Direct Block. Right: Single Indirect Block

  - Triple Indirect; Block number of a block containing block numbers of blocks containing block numbers of blocks containing block numbers

- Inode Summary
  - The inode contains the on disk data associated with a file
  - Contains mode, owner, and other bookkeeping
  - Efficient random and sequential access via indexed allocation
  - Small files (the majority of files) require only a single access
  - Larger files require progressively more disk accesses for random access; Sequential access is still efficient
  - Can support really large files via increasing levels of indirection
- **Where/How are Inodes Stored**
- **System V Disk Layout (s5fs)** (see Fig. 6.21Upper)
  - Boot Block; contain code to bootstrap the OS
  - Super Block; Contains attributes of the file system itself; e.g. size, number of inodes, start block of inode array, start of data block area, free inode list, free data block list
  - Inode Array
  - Data blocks
- Some problems with s5fs

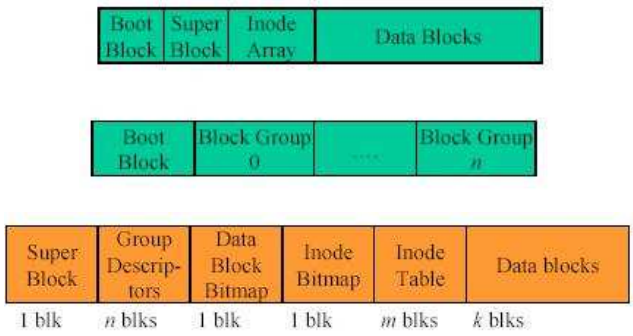


Figure 6.21: Upper: System V Disk Layout (s5fs). Middle: Layout of an Ext2 Partition. Lower: Layout of a Block Group.

- Inodes at start of disk; data blocks end. Long seek times; Must read inode before reading data blocks
- Only one superblock; Corrupt the superblock and entire file system is lost
- Block allocation suboptimal; Consecutive free block list created at FS format time. Allocation and deallocation eventually randomizes the list resulting the random allocation
- Inodes allocated randomly; Directory listing resulted in random inode access patterns

- **The Linux Ext2 File System** (see Fig. 6.21Middle)
  - Second Extended Filesystem; Evolved from Minix filesystem (via "Extended Filesystem")
  - Features
    - Block size (1024, 2048, and 4096) configured as FS creation
    - Preallocated inodes (max number also configured at FS creation)
    - Block groups to increase locality of reference
    - Symbolic links ¡ 60 characters stored within inode
  - Main Problem: unclean unmount → **e2fsck**
    - Ext3fs keeps a journal of (metadata) updates
    - Journal is a file where updated are logged
    - Compatible with ext2fs

- Layout of an Ext2 Partition
  - Disk divided into one or more partitions
  - Partition:
    - Reserved boot block,
    - Collection of equally sized block groups,
    - All block groups have the same structure
- **Layout of a Block Group** (see Fig. 6.21Lower)
  - Replicated super block and group descriptors; For e2fsck
  - Bitmaps identify used inodes/blocks
  - All block have the same number of data blocks
  - Advantages of this structure:
    - Replication simplifies recovery
    - Proximity of inode tables and data blocks (reduces seek time)

6.5 vita

Cem Özdoğan was born in Merzifon, Amasya on October 23, 1969. He received his B.S. degree in Physics from the Middle East Technical University in June 1994. He received his M.S. degree in Physics from the Middle East Technical University in June 1996. He received his Ph.D. degree in Physics from the Middle East Technical University in June 2002. He worked as a research assistant from 1994 to 2001 in the department of physics, Kırıkkale University and Middle East Technical University and as instructor in the department of computer engineering, Çankaya University from 2001 to 2002. He is currently employed as Assist. Prof. in the department of computer engineering, Çankaya University. His main areas of interest are electronic structure calculations, parallel computing and scientific computing.