1

INTRODUCTION TO OPERATING SYSTEMS

Unit Structure

1.0 OBJECTIVES

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

- Understand the fundamental concepts and techniques of operating systems.
- Build a core knowledge of what makes an operating system tick.

- Identify various classes of operating systems and distinguish between them.

1.1 INTRODUCTION

Each user has his own personal thoughts on what the computer system is for. The operating system, or OS, as we will often call it, is the intermediary between users and the computer system. It provides the services and features present in abstract views of all its users through the computer system.

An operating system controls use of a computer system's resources such as CPUs, memory, and I/O devices to meet computational requirements of its users.

1.2 OS AND COMPUTER SYSTEM

In technical language, we would say that an individual user has an abstract view of the computer system, a view that takes in only those features that the user considers important. To be more specific, typical hardware facilities for which the operating system provides abstractions include:

- processors
- RAM (random-access memory, sometimes known as primary storage, primary memory, or physical memory)
- disks (a particular kind of secondary storage)
- network interface
- display
- keyboard
- mouse

An operating system can also be commonly defined as "*a program running at all times on the computer (usually called the kernel), with all other being application programs*".

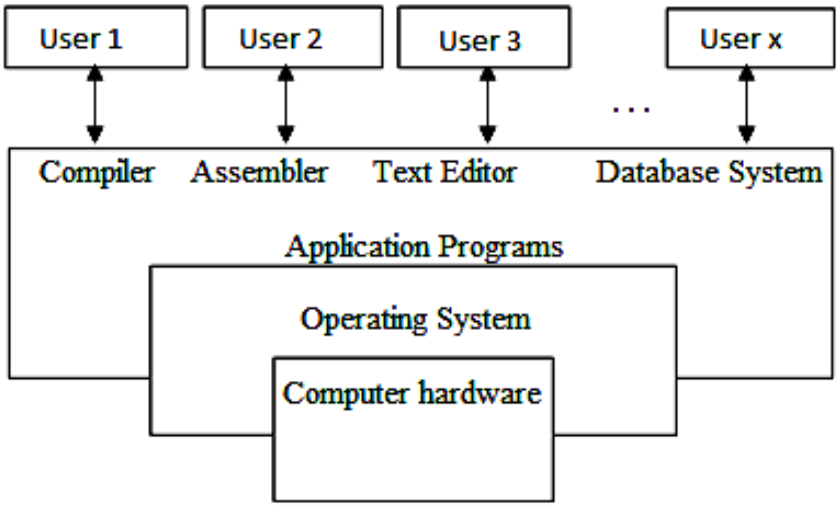


Fig. 1.1 An abstract view of the components of an Operating System

A computer system can be divided roughly into four components: *the hardware, the operating system, the application programs and the users*.

1.3 SYSTEM PERFORMANCE

A modern OS can service several user programs simultaneously. The OS achieves it by interacting with the computer and user programs to perform several control functions.

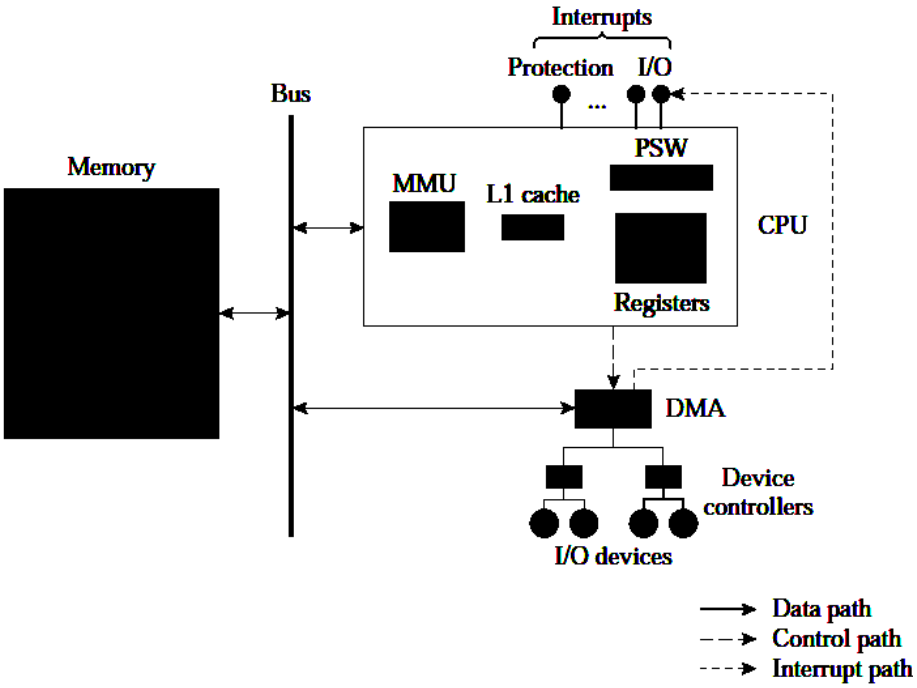


Fig 1.2 Schematic of a computer

The CPU contains a set of control registers whose contents govern its functioning. The program status word (PSW) is the collection of control registers of the CPU; we refer to each control register as a field of the PSW. A program whose execution was interrupted should be resumed at a later time. To facilitate this, the kernel saves the CPU state when an interrupt occurs.

The CPU state consists of the PSW and program-accessible registers, which we call general-purpose registers (GPRs). Operation of the interrupted program is resumed by loading back the saved CPU state into the PSW and GPRs.

The input-output system is the slowest unit of a computer; the CPU can execute millions of instructions in the amount of time required to perform an I/O operation. Some methods of performing an I/O operation require participation of the CPU, which wastes valuable CPU time.

Hence the input-output system of a computer uses direct memory access (DMA) technology to permit the CPU and the I/O system to operate independently. The operating system exploits this feature to let the CPU execute instructions in a program while I/O operations of the same or different programs are in progress. This technique reduces CPU idle time and improves system performance.

1.4 CLASSES OF OPERATING SYSTEMS

Classes of operating systems have evolved over time as computer systems and users' expectations of them have developed; i.e., as computing environments have evolved.

Table 1.1 lists eight fundamental classes of operating systems that are named according to their defining features. The table shows when operating systems of each class first came into widespread use; what fundamental effectiveness criterion, or prime concern, motivated its development; and what key concepts were developed to address that prime concern.

OS Class	Period	Prime Concern	Key Concepts
Batch Processing Systems	1960s	CPU idle time	Automate transition between jobs
Time Sharing Systems	1970s	Good response time	Time-slice, round-robin scheduling
Multiprocessing Systems	1980s	Master/Slave processor priority	Symmetric/Asymmetric multiprocessing
Real Time Systems	1980s	Meeting time constraints	Real-time scheduling
Distributed Systems	1990s	Resource sharing	Distributed control, transparency
Desktop Systems	1970s	Good support to a single user	Word processing, Internet access
Handheld Systems	Late 1980s	32-bit CPUs with protected mode	Handle telephony, digital photography, and third party applications
Clustered Systems	Early 1980s	Low cost µps, high speed networks	Task scheduling, node failure management

TABLE 1.1
KEY FEATURES OF CLASSES OF OPERATING SYSTEMS

1.4.1 BATCH PROCESSING SYSTEMS

To improve utilization, the concept of a batch operating system was developed. It appears that the first batch operating system (and the first OS of any kind) was developed in the mid-

1950s by General Motors for use on an IBM 701 [WEIZ81]. The concept was subsequently refined and implemented on the IBM 704 by a number of IBM customers. By the early 1960s, a number of vendors had developed batch operating systems for their computer systems. IBSYS, the IBM operating system for the 7090/7094 computers, is particularly notable because of its widespread influence on other systems.

In a batch processing operating system, the prime concern is CPU efficiency. The batch processing system operates in a strict one job-at-a-time manner; within a job, it executes the programs one after another. Thus only one program is under execution at any time.

The opportunity to enhance CPU efficiency is limited to efficiently initiating the next program when one program ends, and the next job when one job ends, so that the CPU does not remain idle.

1.4.1.1 SIMPLE BATCH SYSTEMS

With a batch operating system, processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead. Despite this overhead, the simple batch system improves utilization of the computer.

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

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

Fig 1.3 System utilisation example

1.4.1.2 MULTI-PROGRAMMED BATCH SYSTEMS

Multiprogramming operating systems are fairly sophisticated compared to single-program, or uniprogramming, systems. To have several jobs ready to run, they must be kept in main memory, requiring some form of memory management. In addition, if several jobs are ready to run, the processor must decide which one to run, this decision requires an algorithm for scheduling. These concepts are discussed in later chapters.

There must be enough memory to hold the OS (resident monitor) and one user program. Suppose there is room for the OS and two user programs.

When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O (Figure 1.4(b)). Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them (Figure 1.4(c)). The approach is known as multiprogramming, or multitasking. It is the central theme of modern operating systems.

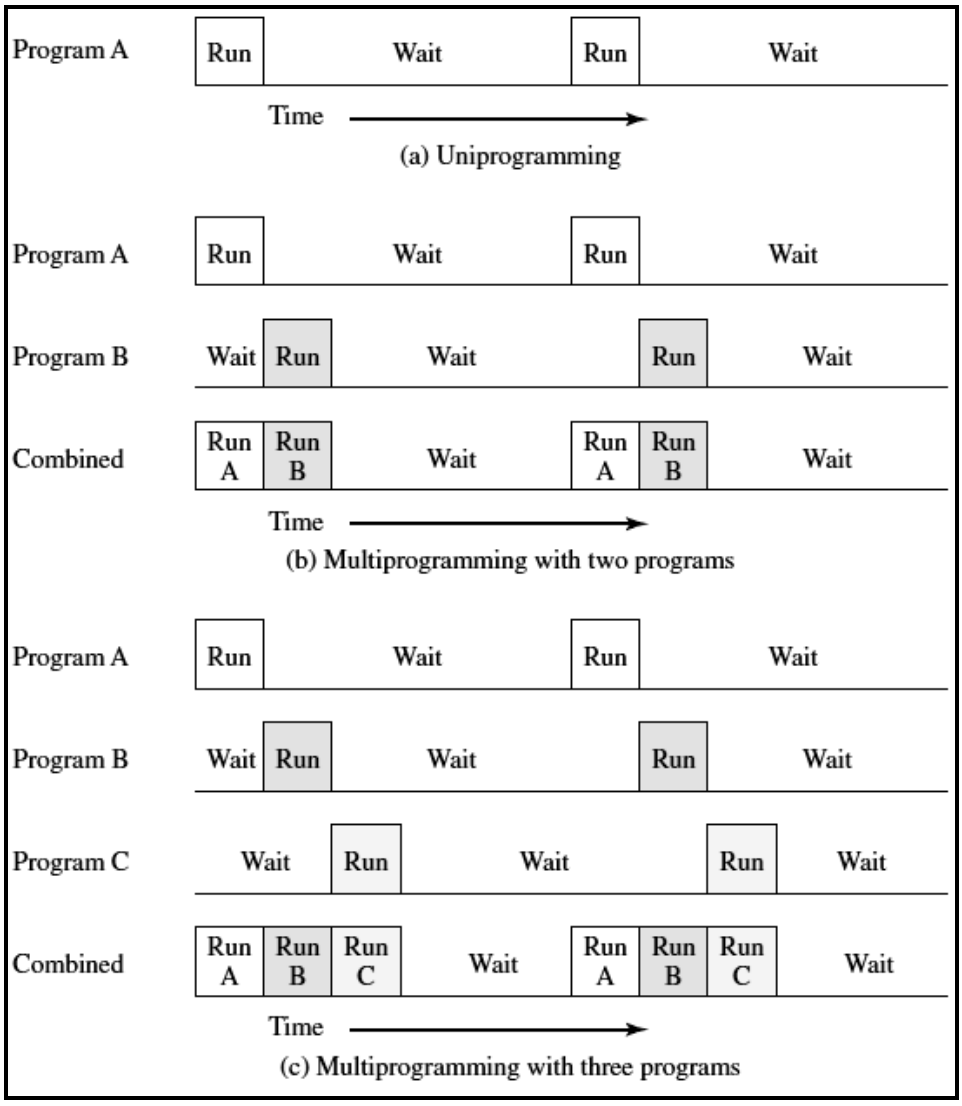


Fig 1.4 Multiprogramming Example

This idea also applies to real life situations. You do not have only one subject to study. Rather, several subjects may be in the process of being served at the same time. Sometimes, before studying one entire subject, you might check some other subject to avoid monotonous study. Thus, if you have enough subjects, you never need to remain idle.

1.4.2 TIME SHARING SYSTEMS

A time-sharing operating system focuses on facilitating quick response to *subrequests* made by all processes, which provides a tangible benefit to users. It is achieved by giving a fair execution opportunity to each process through two means: The OS services all processes by turn, which is called round-robin scheduling. It also prevents a process from using too much CPU time when scheduled to execute, which is called time-slicing. The combination of these two techniques ensures that no process has to wait long for CPU attention.

1.4.3 MULTIPROCESSING SYSTEMS

Many popular operating systems, including Windows and Linux, run on multiprocessors. Multiprocessing sometimes refers to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant. However, the terms multitasking or multiprogramming are more appropriate to describe this concept, which is implemented mostly in software, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs. A system can be both multiprocessing and multiprogramming, only one of the two, or neither of the two.

Systems that treat all CPUs equally are called symmetric multiprocessing (SMP) systems. In systems where all CPUs are not equal, system resources may be divided in a number of ways, including asymmetric multiprocessing (ASMP), non-uniform memory access (NUMA) multiprocessing, and clustered multiprocessing.

1.4.3.1 SYMMETRIC MULTIPROCESSING SYSTEMS

Symmetric multiprocessing or SMP involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory. Most common multiprocessor systems today use an SMP architecture.

In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors. SMP systems allow any processor to work on any task no matter where the data for that task are located in memory; with proper operating system support, SMP systems can easily move tasks between processors to balance the workload efficiently.

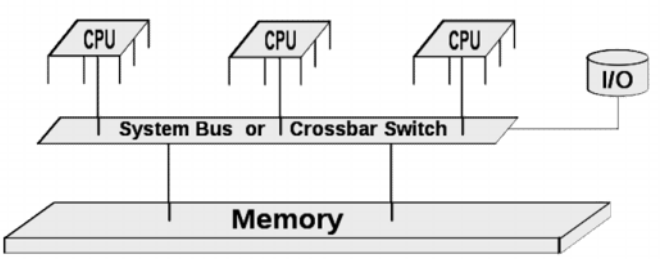


Fig 1.5 A typical SMP system

1.4.3.2 ASYMMETRIC MULTIPROCESSING SYSTEMS

Asymmetric hardware systems commonly dedicated individual processors to specific tasks. For example, one processor may be dedicated to disk operations, another to video operations, and the rest to standard processor tasks .These systems don't have the flexibility to assign processes to the least-loaded CPU, unlike an SMP system.

Unlike SMP applications, which run their threads on multiple processors, ASMP applications will run on one processor but outsource smaller tasks to another. Although the system may physically be an SMP, the software is still able to use it as an ASMP by simply giving certain tasks to one processor and deeming it the "master", and only outsourcing smaller tasks to "slave "processors.

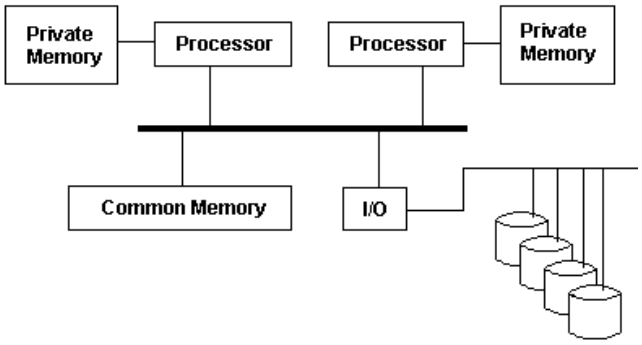


Fig 1.6 Multiple processors with unique access to memory and I/O

1.4.4 REAL TIME SYSTEMS

A real-time operating system is used to implement a computer application for controlling or tracking of real-world activities. The application needs to complete its computational tasks in a timely manner to keep abreast of external events in the activity that it controls. To facilitate this, the OS permits a user to create several processes within an application program, and uses real-time scheduling to interleave the execution of processes such that the application can complete its execution within its time constraint.

1.4.4.1 HARD AND SOFT REAL-TIME SYSTEMS

To take advantage of the features of real-time systems while achieving maximum cost-effectiveness, two kinds of real-time systems have evolved.

A hard real-time system is typically dedicated to processing real-time applications, and provably meets the response requirement of an application under all conditions.

A soft real-time system makes the best effort to meet the response requirement of a real-time application but cannot guarantee that it will be able to meet it under all conditions. Digital audio or multimedia systems fall in this category. Digital telephones are also soft real-time systems.

1.4.4.2 FEATURES OF A REAL-TIME OPERATING SYSTEM

Feature	Explanation
Concurrency within an application	A programmer can indicate that some parts of an application should be executed concurrently with one another. The OS considers execution of each such part as a process.
Process priorities	A programmer can assign priorities to processes.
Scheduling	The OS uses priority-based or deadline-aware scheduling.
Domain-specific events, interrupts	A programmer can define special situations within the external system as events, associate interrupts with them, and specify event handling actions for them.
Predictability	Policies and overhead of the OS should be predictable.
Reliability	The OS ensures that an application can continue to function even when faults occur in the computer.

1.4.5 DISTRIBUTED SYSTEMS

A distributed operating system permits a user to access resources located in other computer systems conveniently and reliably. To enhance convenience, it does not expect a user to know the location of resources in the system, which is called transparency. To enhance efficiency, it may execute parts of a computation in different computer systems at the same time. It uses distributed control; i.e., it spreads its decision-making actions

across different computers in the system so that failures of individual computers or the network does not cripple its operation.

A distributed operating system is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. The users may not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in certain critical ways. Distributed systems, for example, often allow applications to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism.

1.4.6 DESKTOP SYSTEMS

A desktop system is a personal computer (PC) system in a form intended for regular use at a single location, as opposed to a mobile laptop or portable computer. Early desktop computers were designed to lay flat on the desk, while modern towers stand upright. Most modern desktop computer systems have separate screens and keyboards.

Modern ones all support multiprogramming, often with dozens of programs started up at boot time. Their job is to provide good support to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Linux, FreeBSD, Windows 8, and the Macintosh operating system. Personal computer operating systems are so widely known that probably little introduction is needed.

1.4.7 HANDHELD SYSTEMS

A handheld computer or PDA (Personal Digital Assistant) is a small computer that fits in a shirt pocket and performs a small number of functions, such as an electronic address book and memo pad. Since these computers can be easily fitted on the palmtop, they are also known as palmtop computers. Furthermore, many mobile phones are hardly any different from PDAs except for the keyboard and screen. In effect, PDAs and mobile phones have essentially merged, differing mostly in size, weight, and user interface. Almost all of them are based on 32-bit CPUs with protected mode and run a sophisticated operating system.

One major difference between handhelds and PCs is that the former do not have multigigabyte hard disks, which changes a lot.

Two of the most popular operating systems for handhelds are Symbian OS and Android OS.

1.4.8 CLUSTERED SYSTEMS

A computer cluster consists of a set of loosely connected computers that work together so that in many respects they can be viewed as a single system.

The components of a cluster are usually connected to each other through fast local area networks, each node (computer used as a server) running its own instance of an operating system. Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high speed networks, and software for high performance distributed computing.

In Clustered systems, if the monitored machine fails, the monitoring machine can take ownership of its storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down, but the users and clients of the application would only see a brief interruption of the service.

In asymmetric clustering, one machine is in hot standby mode while the other is running the applications. The hot standby host (machine) does nothing but monitor the active server. If that server fails, the hot standby host becomes the active server. In symmetric mode, two or more hosts are running applications, and they are monitoring each other. It does require that more than one application be available to run.

Other forms of clusters include parallel clusters and clustering over a WAN. Parallel clusters allow multiple hosts to access the same data on the shared storage and are usually accomplished by special version of software and special releases of applications. For example, Oracle Parallel Server is a version of Oracle's database that has been designed to run parallel clusters. Storage-area networks (SANs) are the feature development of the clustered systems includes the multiple hosts to multiple storage units.

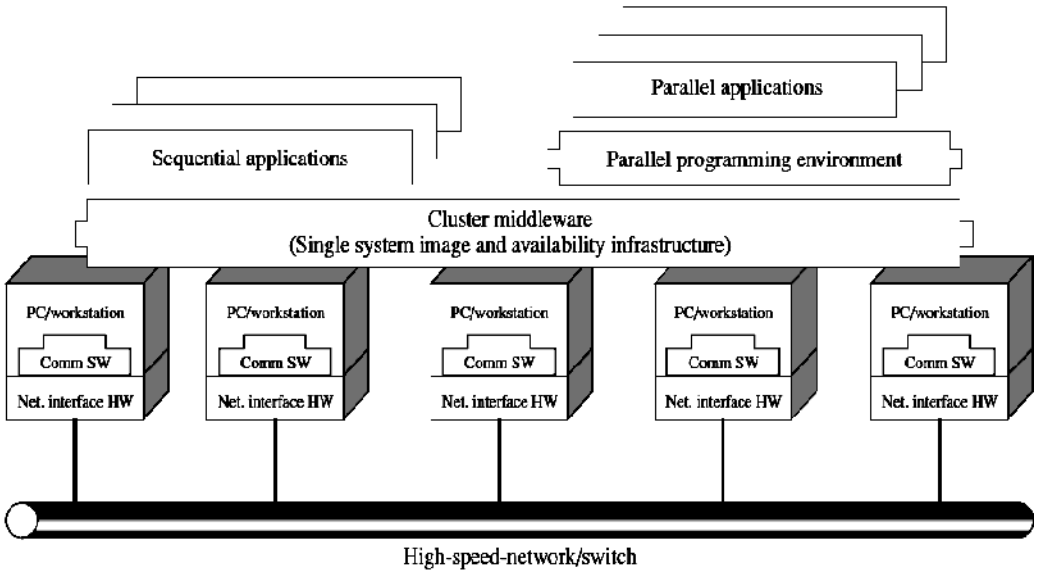


Fig 1.7 Cluster Computer Architecture

1.5 LET US SUM UP

- The batch processing system operates in a strict one job-at-a-time manner; within a job, it executes the programs one after another.

- A time-sharing operating system focuses on facilitating quick response to *subrequests* made by all processes, which provides a tangible benefit to users.

- Systems that treat all CPUs equally are called symmetric multiprocessing (SMP) systems.

- In systems where all CPUs are not equal, system resources may be divided in a number of ways, including asymmetric multiprocessing (ASMP), non-uniform memory access (NUMA) multiprocessing, and clustered multiprocessing.

- A hard real-time system is typically dedicated to processing real-time applications, and provably meets the response requirement of an application under all conditions.

- A soft real-time system makes the best effort to meet the response requirement of a real-time application but cannot guarantee that it will be able to meet it under all conditions.

- A distributed operating system is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors.

- A desktop system is a personal computer (PC) system in a form intended for regular use at a single location, as opposed to a mobile laptop or portable computer.

- One major difference between handhelds and PCs is that the former do not have multigigabyte hard disks, which changes a lot.

- Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high speed networks, and software for high performance distributed computing.

1.6 UNIT END QUESTIONS

1. State the various classes of an operating system.
2. What are the differences between symmetric and asymmetric multiprocessing system?
3. Briefly explain Real-Time Systems.
4. Write a note on Clustered Systems.
5. What are the key features of classes of operating systems?

2

TRANSFORMATION & EXECUTION OF PROGRAMS

Unit Structure

2.0 OBJECTIVES

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

- Study the transformation and execution of a program.

2.1 INTRODUCTION

An operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, and command interpreters are not part of the operating system, even though they are important and useful. But still we study them as they use many OS resources.

2.2 TRANSLATORS AND COMPILERS

A translator is a program that takes a program written in one programming language (the source language) as input and produces a program in another language (the object or target language) as output.

If the source language is a high-level language such as FORTRAN (FORmula TRANslator), PL/I, or COBOL, and the object language is a low-level language such as an assembly language (machine language), then such a translator is called a compiler.

Executing a program written in a high-level programming language is basically a two-step process. The source program must first be compiled i.e. translated into the object program. Then the resulting object program is loaded into memory and executed.

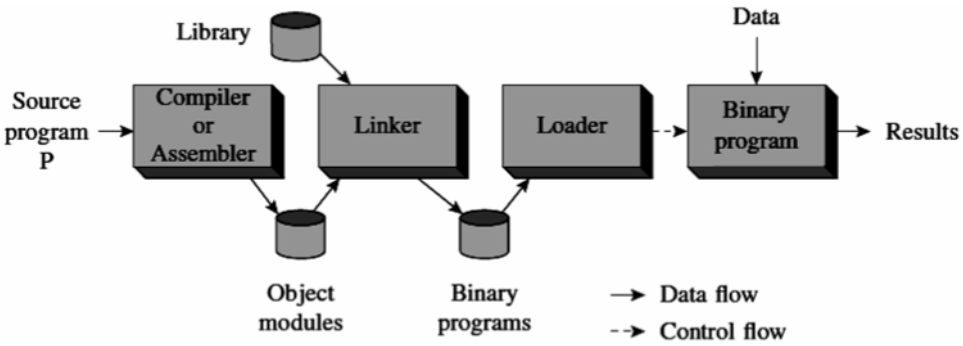


Fig 2.1 Schematic diagram of transformation and execution of a program

Compilers were once considered almost impossible programs to write. The first FORTRAN compiler, for example, took 18 man-years to implement (Backus et al. [1957]). Today, however, compilers can be built with much less effort. In fact, it is not unreasonable to expect a fairly substantial compiler to be implemented as a student project in a one-semester compiler design course. The principal developments of the past twenty years which led to this improvement are:

• The understanding of organizing and modularizing the process of compilation.

• The discovery of systematic techniques for handling many of the important tasks that occur during compilation.

• The development of software tools that facilitate the implementation of compilers and compiler components.

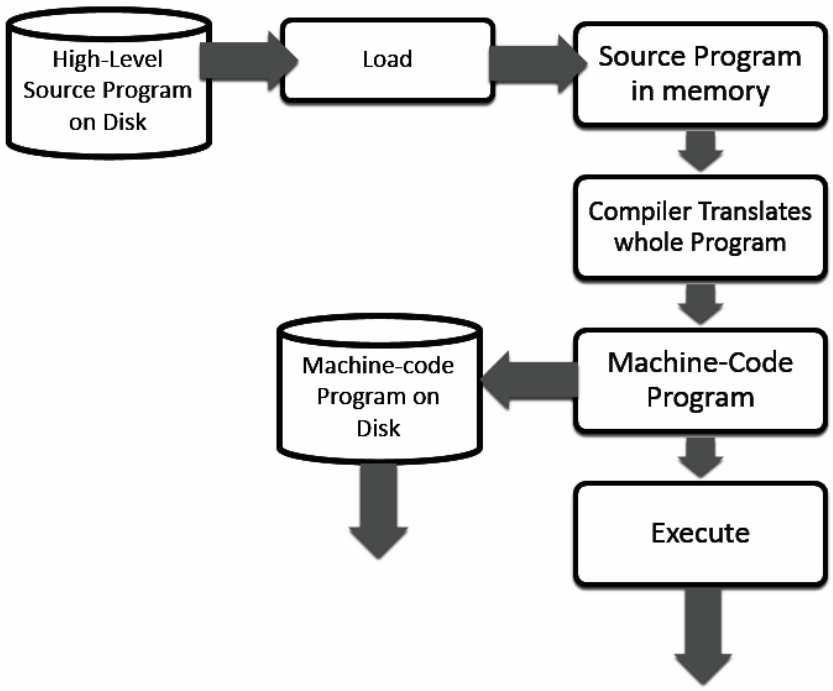


Fig 2.2 Working of a Compiler

2.3 ASSEMBLERS

Assembly language is a type of low-level language and a program that compiles it is more commonly known as an *assembler*, with the inverse program known as a *disassembler*.

The assembler program recognizes the character strings that make up the symbolic names of the various machine operations, and substitutes the required machine code for each instruction. At the same time, it also calculates the required address in memory for each symbolic name of a memory location, and substitutes those addresses for the names resulting in a machine language program that can run on its own at any time.

In short, an assembler converts the assembly codes into binary codes and then it assembles the machine understandable code into the main memory of the computer, making it ready for execution.

The original assembly language program is also known as the source code, while the final machine language program is designated the object code. If an assembly language program needs to be changed or corrected, it is necessary to make the changes to the source code and then re-assemble it to create a new object program.

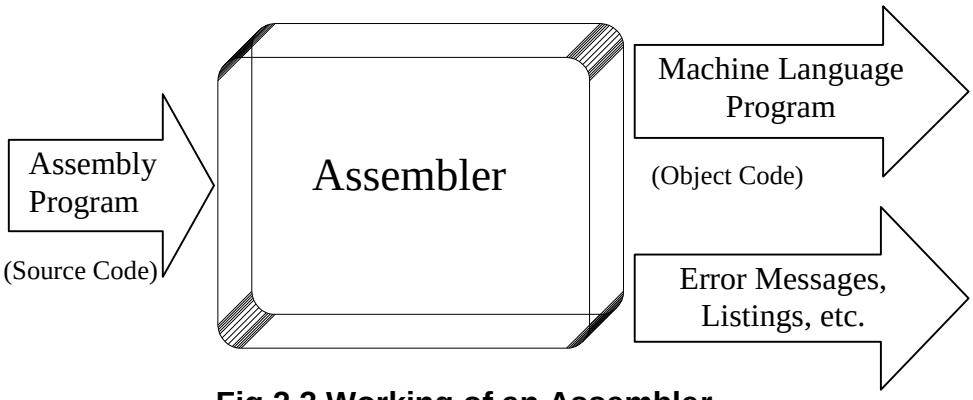


Fig 2.3 Working of an Assembler

The functions of an assembler are given below:

- It allows the programmer to use mnemonics while writing source code programs, which are easier to read and follow.
- It allows the variables to be represented by symbolic names, not as memory locations.
- It translates mnemonic operations codes to machine code and corresponding register addresses to system addresses.
- It checks the syntax of the assembly program and generates diagnostic messages on syntax errors.
- It assembles all the instructions in the main memory for execution.
- In case of large assembly programs, it also provides linking facility among the subroutines.
- It facilitates the generation of output on required output medium.

2.4 INTERPRETERS

Unlike compilers, an interpreter translates a statement in a program and executes the statement immediately, before translating the next source language statement. When an error is encountered in the program, the execution of the program is halted and an error message is displayed. Similar to compilers, every interpreted language such as BASIC and LISP have their own interpreters.

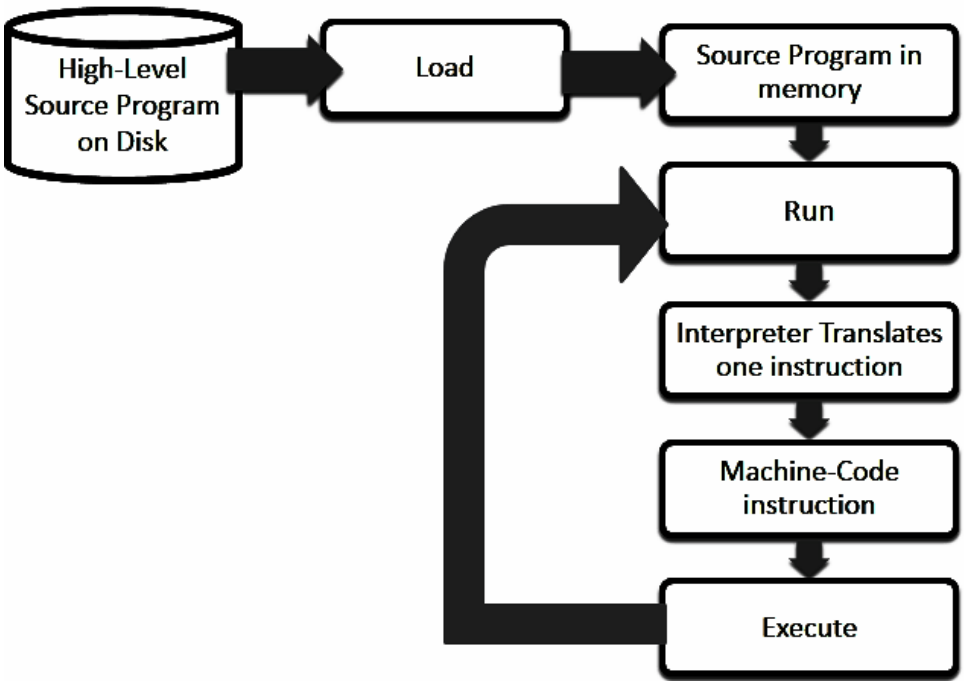


Fig 2.4 Working of an Interpreter

We may think of the intermediate code as the machine language of an abstract computer designed to execute the source code. For example, SNOBOL is often interpreted, the intermediate code being a language called Polish postfix notation.

In some cases, the source language itself can be the intermediate language. For example, most command languages, such as JCL, in which one communicates directly with the operating system, are interpreted with no prior translation at all.

Interpreters are often smaller than compilers and facilitate the implementation of complex programming language constructs. However, the main disadvantage of interpreters is that the execution time of an interpreted program is usually slower than that of a corresponding compiled object program.

2.4.1 COMPILED VERSUS INTERPRETED LANGUAGES

Higher-level programming languages usually appear with a type of translation in mind: either designed as compiled language or interpreted language. However, in practice there is rarely anything about a language that requires it to be exclusively compiled or exclusively interpreted, although it is possible to design languages that rely on re-interpretation at run time. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Interpretation does not replace compilation completely. It only hides it from the user and makes it gradual. Even though an interpreter can itself be interpreted, a directly executed program is needed somewhere at the bottom of the stack (see machine language). Modern trends toward just-in-time compilation and bytecode interpretation at times blur the traditional categorizations of compilers and interpreters.

Some language specifications spell out that implementations must include a compilation facility; for example, Common Lisp. However, there is nothing inherent in the definition of Common Lisp that stops it from being interpreted. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programs to construct arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

2.5 LINKERS

An application usually consists of hundreds or thousands of lines of codes. The codes are divided into logical groups and stored in different modules so that the debugging and maintenance of the codes becomes easier. Hence, for an application, it is always advisable to adhere to structural (modular) programming practices. When a program is broken into several modules, each module can be modified and compiled independently. In such a case, these modules have to be linked together to create a complete application. This job is done by a tool known as linker.

A linker is a program that links several object modules and libraries to form a single, coherent program (executable). Object modules are the machine code output from an assembler or compiler and contain executable machine code and data, together with information that allows the linker to combine the modules together to form a program.

Generally, all high-level languages use some in-built functions like calculating square roots, finding logarithmic values, and so on. These functions are usually provided by the language itself, the programmer does not need to code them separately. During the program execution process, when a program invokes any in-built function, the linker transfers the control to that program where the function is defined, by making the addresses of these functions known to the *calling* program.

The various components of a process are illustrated in Fig. 2.5 for a program with three C files and two header files.

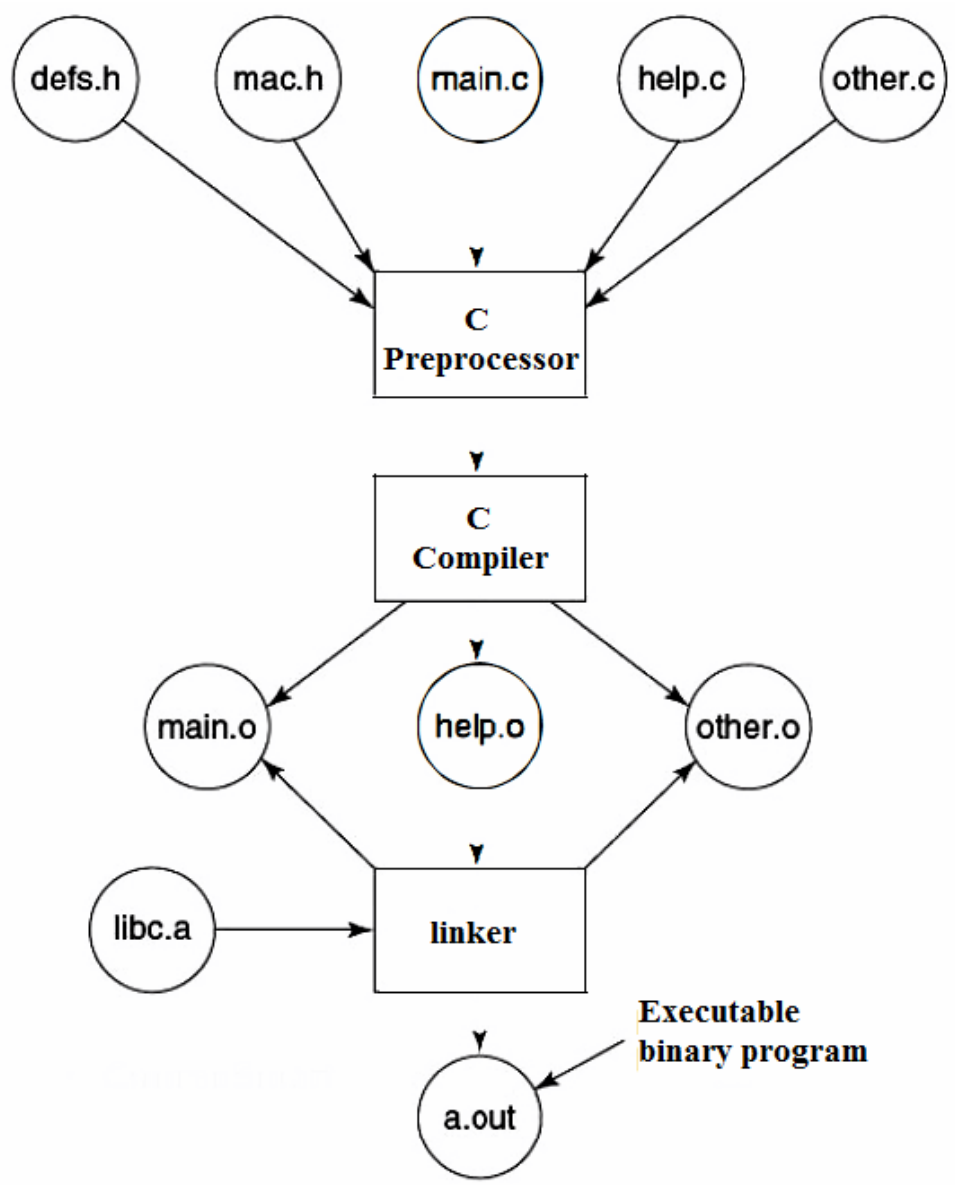


Fig 2.5 The process of compiling C and header files to make an executable file

The addresses assigned by *linker*s are called linked addresses. The user may specify the linked origin for the program; otherwise, the linker assumes the linked origin to be the same as the translated origin. In accordance with the linked origin and the relocation necessary to avoid address conflicts, the linker binds instructions and data of the program to a set of linked addresses. The resulting program, which is in a ready-to-execute program form called a binary program, is stored in a library. The directory of the library stores its name, linked origin, size, and the linked start address.

2.6 LET US SUM UP

- A translator is a program that takes a program written in one programming language (the source language) as input and produces a program in another language (the object or target language) as output.

- If the source language is a high-level language such as FORTRAN (FORmula TRANslator), PL/I, or COBOL, and the object language is a low-level language such as an assembly language (machine language), then such a translator is called a compiler.

- An assembler converts the assembly codes into binary codes and then it assembles the machine understandable code into the main memory of the computer, making it ready for execution.

- An interpreter translates a statement in a program and executes the statement immediately, before translating the next source language statement.

- A linker is a program that links several object modules and libraries to form a single, coherent program (executable).

2.7 UNIT END QUESTIONS

1. Define :
    a. Translator
    b. Assembler
    c. Compiler
    d. Interpreter
    e. Linker
2. State the functions of an assembler.
3. Briefly explain the working of an interpreter.
4. Distinguish between Compiled versus interpreted Languages.
5. What is a linker? Explain with the help of a diagram.

3

OS SERVICES, CALLS, INTERFACES AND PROGRAMS

Unit Structure

3.0 OBJECTIVES

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

- Study different OS Services
- Study different System Calls

3.1 INTRODUCTION

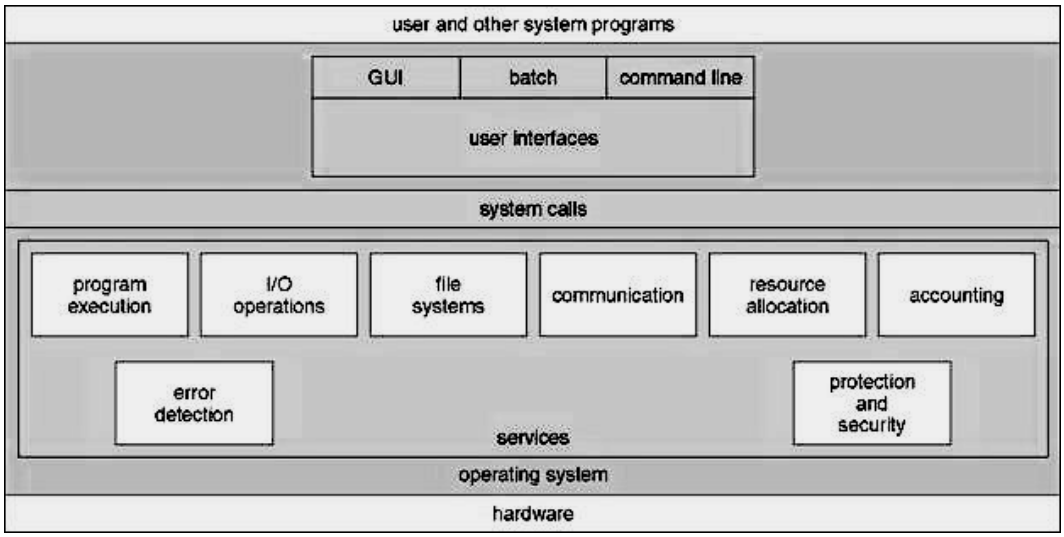


Fig 3.1 A view of Operating System Services

3.2 OPERATING SYSTEM SERVICES

The Operating –System Services are provided for the convenience of the programmer, to make the programming task easier. One set of operating-system services provides functions that are helpful to the user:

3.2.1 PROGRAM EXECUTION:

The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally forcefully (using notification).

3.2.2 I/O OPERATION:

I/O may involve a file or an I/O device. Special functions may be desired (such as to rewind a tape drive, or to blank a CRT screen). I/O devices are controlled by O.S.

3.2.3 FILE-SYSTEMS:

File system program reads, writes, creates and deletes files by name.

3.2.4 COMMUNICATIONS:

In many circumstances, one process needs to exchange information with another process. Communication may be implemented via shared memory, or by the technique of message passing, in which packets of information are moved between processes by the O.S..

3.2.5 RESOURCE ALLOCATION:

When multiple users are logged on the system or multiple jobs are running at the same time, resources such as CPU cycles, main memory, and file storage etc. must be allocated to each of them. O.S. has CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There are routines for tape drives, plotters, modems, and other peripheral devices.

3.2.6 ACCOUNTING:

To keep track of which user uses how many and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.

3.2.7 ERROR DETECTION:

Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or vast use of CPU time). O.S should take an appropriate action to resolve these errors.

3.2.8 PROTECTION AND SECURITY:

The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other

- **Protection** involves ensuring that all access to system resources is controlled.

- **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.

- If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

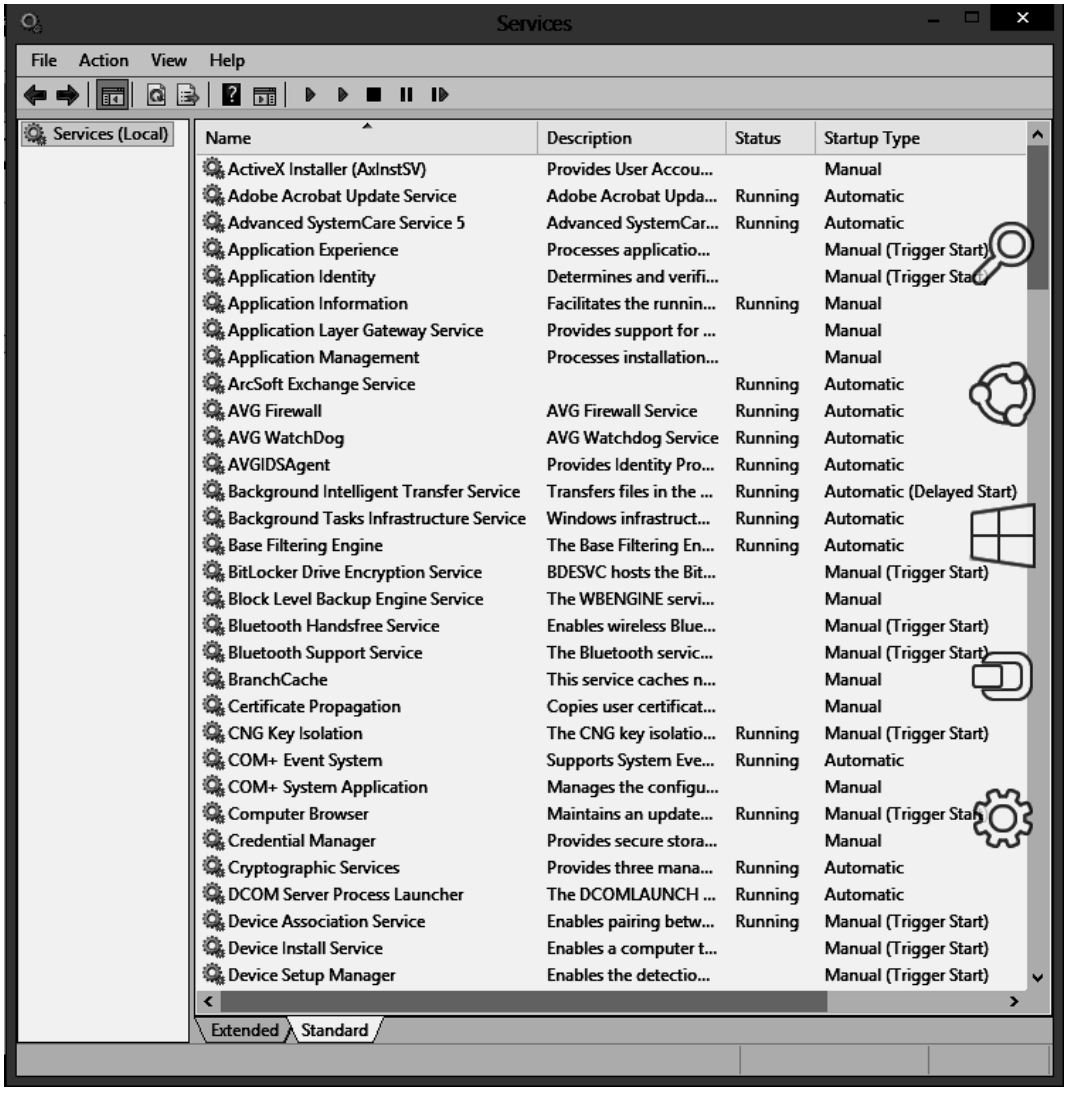


Fig 3.2 Microsoft Windows 8 Operating System Services

3.3 USER OPERATING SYSTEM INTERFACE

Almost all operating systems have a user interface (UI) varying between Command-Line Interface (CLI) and Graphical User Interface (GUI). These services differ from one operating system to another but they have some common classes.

3.3.1 Command Interpreter:

It is the interface between user and OS. Some O.S. includes the command interpreter in the kernel. Other O.S., such as MS-DOS and UNIX, treat the command interpreter as a special program that is running when a job is initiated, or when a user first logs on (on time-sharing systems). This program is sometimes called ***the control-card interpreter*** or the ***command-line interpreter***, and is often known as the shell. Its function is simple: To get the next command statement and execute it. The command statements themselves deal with process creation and management, I/O handling, secondary storage management, main-memory management, file –system access, protection, and networking. The MS-DOS and UNIX shells operate in this way.

3.3.2 Graphical User Interface (GUI):

With the development in chip designing technology, computer hardware became quicker and cheaper, which led to the birth of GUI based operating system. These operating systems provide users with pictures rather than just characters to interact with the machine.

A GUI:

- Usually uses mouse, keyboard, and monitor.
- Icons represent files, programs, actions, etc.
- Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
- Invented at Xerox PARC.

Many systems now include both CLI and GUI interfaces

- Microsoft Windows is GUI with CLI “command” shell.
- Apple Mac OS X as “LION” GUI interface with UNIX kernel underneath and shells available.
- Solaris is CLI with optional GUI interfaces (Java Desktop, KDE).

3.4 SYSTEM CALLS

A system call is a request that a program makes to the kernel through a software interrupt.

System calls provide the interface between a process and the operating system. These calls are generally available as assembly-language instructions.

Certain systems allow system calls to be made directly from a high-level language program, in which case the calls normally resemble predefined function or subroutine calls.

Several languages-such as C, C++, and Perl-have been defined to replace assembly language for system programming. These languages allow system calls to be made directly. E.g., UNIX system calls may be invoked directly from a C or C++ program. System calls for modern Microsoft Windows platforms are part of the Win32 application programmer interface (API), which is available for use by all the compilers written for Microsoft Windows.

Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM).

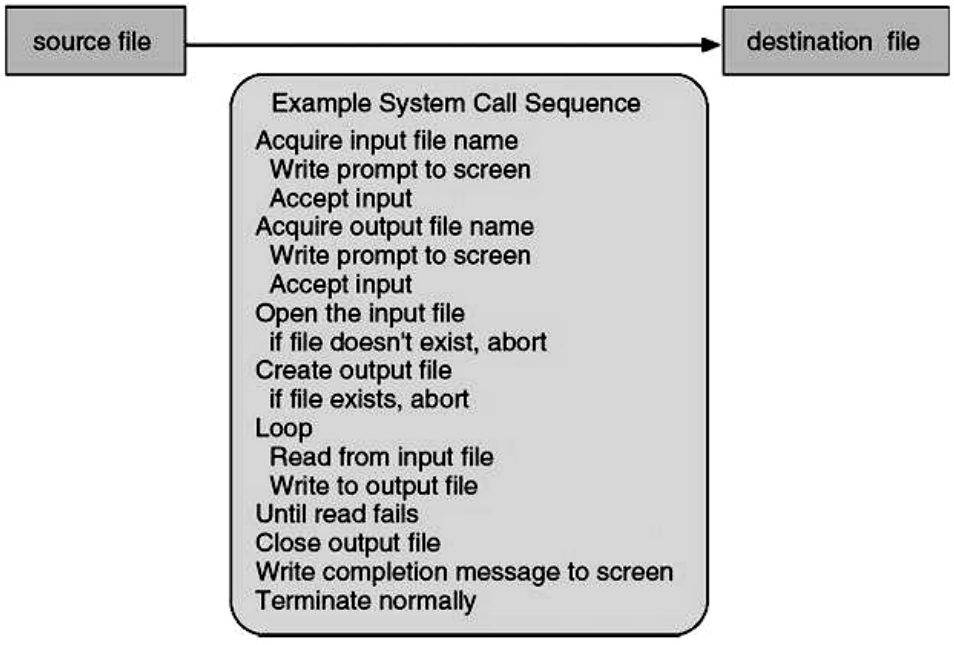


Fig 3.3 Example of System Calls

Call number	Call name	Description
1	exit	Terminate execution of this program

3	read	Read data from a file
4	write	Write data into a file
5	open	Open a file
6	close	Close a file
7	waitpid	Wait for a program's execution to terminate
11	execve	Execute a program
12	chdir	Change working directory
14	chmod	Change file permissions
39	mkdir	Make a new directory
74	sethostname	Set hostname of the computer system
78	gettimeofday	Get time of day
79	settimeofday	Set time of day

Table 3.1 Some Linux System Calls

3.4.1 TYPES OF SYSTEM CALLS:

Traditionally, System Calls can be categorized in six groups, which are: Process Control, File Management, Device Management, Information Maintenance, Communications and Protection.

3.4.1.1 PROCESS CONTROL

End, abort
Load, execute
Create process, terminate process
Get process attributes, set process attributes
Wait for time
Wait event, signal event
Allocate and free memory

3.4.1.2 FILE MANAGEMENT

Create, delete file
Open, close
Read, write, reposition
Get file attributes, set file attributes

3.4.1.3 DEVICE MANAGEMENT

Request device, release device
Read, write, reposition
Get device attributes, set device attributes
Logically attach or detach devices

3.4.1.4 INFORMATION MAINTENANCE

Get time or date, set time or date
Get system data, set system data
Get process, file, or device attributes
Set process, file, or device attributes

3.4.1.5 COMMUNICATIONS

Create, delete communication connection
Send, receive messages
Transfer status information
Attach or detach remote devices

3.4.1.6 PROTECTION

Get File Security, Set File Security
Get Security Group, Set Security Group

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Table 3.2 Examples of Windows and UNIX System Calls

3.5 SYSTEM PROGRAMS

System programs provide a convenient environment for program development and execution. System programs, also known as system utilities, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex. They can be divided into these categories:

3.5.1 FILE MANAGEMENT:

These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

.

3.5.2 STATUS INFORMATION:

Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry which is used to store and retrieve configuration information.

3.5.3 FILE MODIFICATION:

Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

3.5.4 PROGRAMMING-LANGUAGE SUPPORT:

Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.

3.5.5 PROGRAM LOADING AND EXECUTION:

Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, re-locatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.

3.5.6 COMMUNICATIONS:

These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another’s screens, to browse Web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

3.5.7 APPLICATION PROGRAMS:

In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations. Such applications include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical analysis packages, and games.

3.6 OS DESIGN AND IMPLEMENTATION

We face problems in designing and implementing an operating system. There are few approaches that have proved successful.

Design Goals

Specifying and designing an operating system is a highly creative task. The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time shared, single user, multiuser, distributed, real time, or general purpose. Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: *user goals* and *system goals*.

Users desire certain properties in a system. The system should be convenient to use, easy to learn and to use, reliable, safe, and fast. These specifications are not particularly useful in the system design, since there is no general agreement to achieve them.

A similar set of requirements can be defined by people who must design, create, maintain, and operate the system. The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient. Again, these requirements are vague and may be interpreted in various ways. There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for VxWorks, a real-time operating system for embedded systems, must have been substantially different from those for MVS, a large multiuser, multi-access operating system for IBM mainframes.

Implementation

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in higher-level languages such as C or C++. The first system that was not written in assembly language was probably the

Master Control Program (MCP) for Burroughs computers and it was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in PL/1. The Linux and Windows XP operating systems are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

The advantages of using a higher-level language, or at least a systems implementation language, for implementing operating systems are the same as those accrued when the language is used for application programs: the code can be written faster, is more compact, and is easier to understand and debug.

In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to port-to move to some other hardware-if it is written in a higher-level language. For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it runs natively only on the Intel X86 family of CPUs. (Although MS-DOS runs natively only on Intel X86, emulators of the X86 instruction set allow the operating system to run non-natively slower, with more resource use-on other CPUs are programs that duplicate the functionality of one system with another system.) The Linux operating system, in contrast, is written mostly in C and is available natively on a number of different CPUs, including Intel X86, Sun SPARC, and IBMPowerPC.

The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units that can handle the details of complex dependencies much more easily than can the human mind. Major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code.

In addition, although operating systems are large, only a small amount of the code is critical to high performance; the memory manager and the CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottleneck routines can be identified and can be replaced with assembly-language equivalents.

3.7 LET US SUM UP

- Almost all operating systems have a user interface (UI) varying between Command-Line Interface (CLI) and Graphical User Interface (GUI).
- Microsoft Windows is GUI with CLI “command” shell.
- Apple Mac OS X as “LION” GUI interface with UNIX kernel underneath and shells available.
- Solaris is CLI with optional GUI interfaces (Java Desktop, KDE).
- A system call is a request that a program makes to the kernel through a software interrupt.
- System Calls can be categorized in six groups, which are: Process Control, File Management, Device Management, Information Maintenance, Communications and Protection.
- System programs provide a convenient environment for program development and execution.
- The first system that was not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers and it was written in a variant of ALGOL.
- Modern processors have deep pipelining and multiple functional units that can handle the details of complex dependencies much more easily than can the human mind.

3.8 UNIT END QUESTIONS

1. State different operating system services.
2. Describe different system calls.
3. Describe Command interpreter in brief.
4. Write a short note on Design and implementation of an Operating System.

4

OPERATING SYSTEM STRUCTURES

Unit Structure

4.0 OBJECTIVES

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

- Study how components of OS are interconnected and melded into a kernel.
- Study different Virtual Machines
- Distinguish between different levels of Computer

4.1 INTRODUCTION

For efficient performance and implementation an OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics. These subsystems can then be arranged in various architectural configurations discussed in brief in this unit.

4.2 OPERATING SYSTEM STRUCTURES

A modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.

4.2.1 SIMPLE STRUCTURE:

Microsoft Disk Operating System [MS-DOS]: In MS-DOS, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system to crash when user programs fail.

Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible. Another example of limited structuring is the original UNIX operating system. It consists of two separable parts, the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers. We can view the traditional UNIX operating system as being layered. Everything below the system-call interface and above the physical hardware is the kernel.

The kernel provides the file system, CPU scheduling, memory management, and other operating system functions through system calls. Taken in sum that is an enormous amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain.

4.2.2 LAYERED APPROACH:

In layered approach, the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware, the highest (layer N) is the user interface. An operating system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating system layer say, layer M consists of data structures and a set of routines that can be invoked by higher level layers. Layer M, in turn, can invoke operations on lower level layers.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware to implement its functions.

Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Each layer is implemented with only those operations provided by lower level layers. Each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The major difficulty with the layered approach involves appropriately defining the various layers as a layer can use only lower-level layers. Another problem with layered implementations is they tend to be less efficient than other types. For instance, when a user program executes an I/0 operation, it executes a system call that is trapped to the I/0 layer, which calls the memory management layer which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call; the net result is a system call that takes longer than a non-layered system.

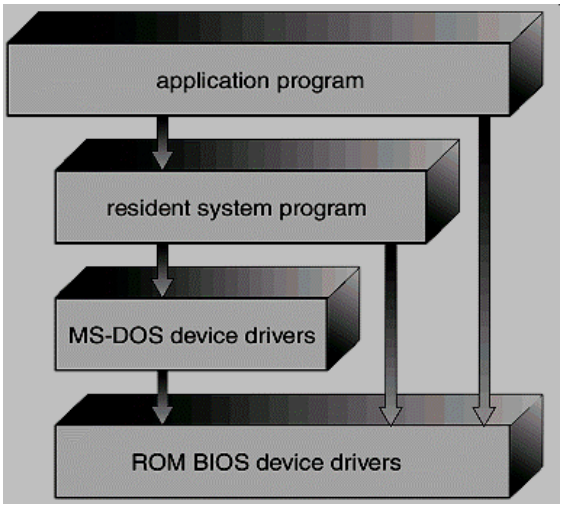


Fig 4.1 MS-DOS LAYER STRUCTURE

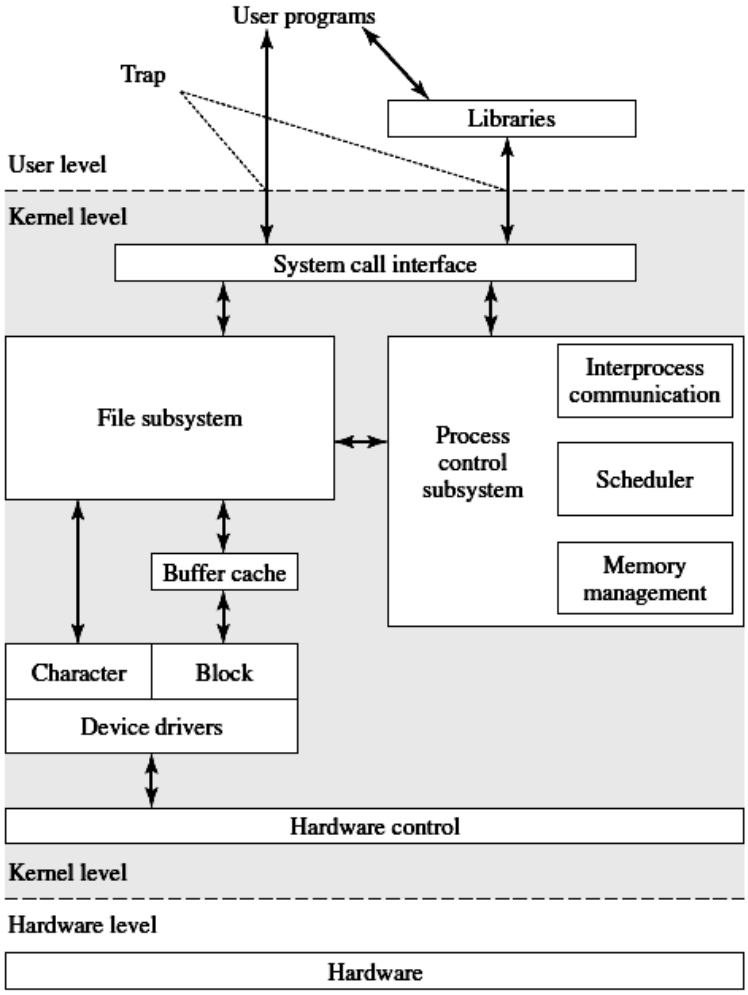


Fig 4.2 Traditional UNIX Kernel

4.2.3 MICROKERNEL APPROACH:

In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach. This method structures the operating system by removing all dispensable components from the kernel and implementing them as system and user level programs. Typically, microkernels provide minimal process and memory management, in addition to a communication facility.

The main function of the micro kernel is to provide a communication facility between the client program and the various services running in user space. One benefit of the microkernel approach is ease of extending the operating system. All new services are added to user space and consequently do not require modification of the kernel. The microkernel also provides more security and reliability, since most services are running as user, rather than kernel-processes. If a service fails, the rest of the operating system remains untouched.

Tru64 UNIX (formerly Digital UNIX) provides a UNIX interface to the user, but it is implemented with a Mach kernel. The Mach

kernel maps UNIX system calls into messages to the appropriate user-level services. The Mac OS X kernel (also known as Darwin) is also based on the Mach micro kernel. Another example is QNX, a real-time operating system. The QNX microkernel provides services for message passing and process scheduling. It also handles low-level network communication and hardware interrupts. All other services in QNX are provided by standard processes that run outside the kernel in user mode.

Microkernels can suffer from decreased performance due to increased system function overhead.

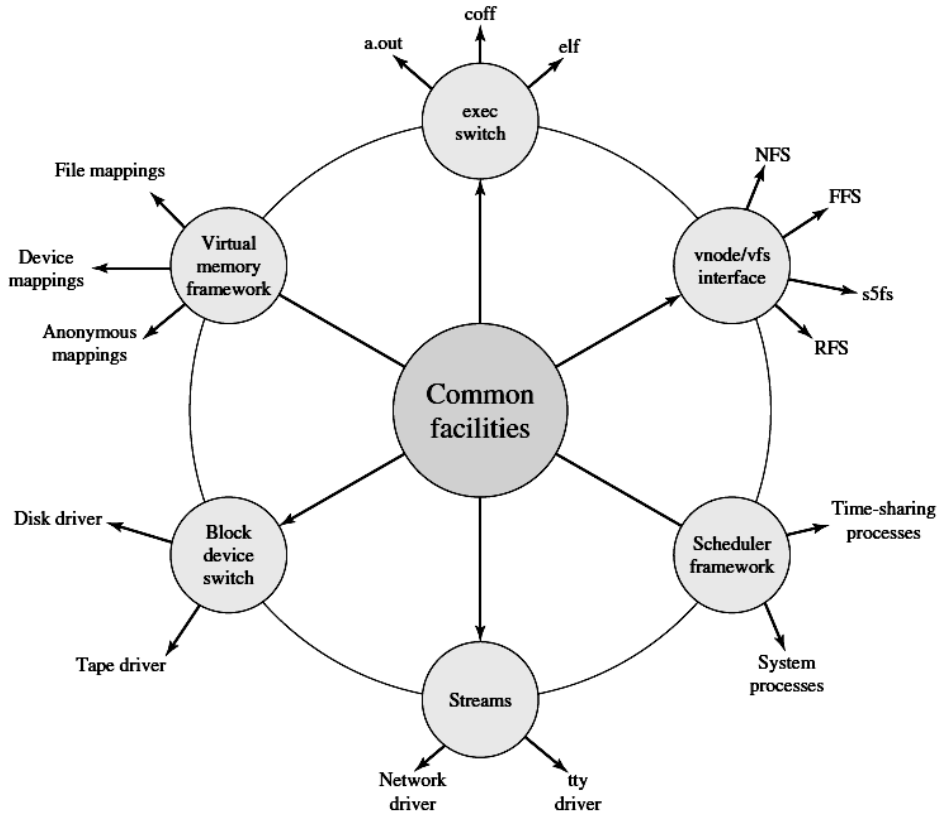


Fig 4.3 Modern UNIX Kernel

4.2.4 MODULES:

The current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel. Here, the kernel has a set of core components and links in additional services either during boot time or during run time. Such a strategy uses dynamically loadable modules. Most current UNIX-like systems, and Microsoft Windows, support loadable kernel modules, although they might use a different name for them, such as kernel loadable module (*kld*) in FreeBSD and kernel extension (*kext*) in OS X. They are also known as Kernel Loadable Modules (or *KLM*), and simply as Kernel Modules (*KMOD*). For example, the Solaris operating system structure, shown in figure

below, is organized around a core kernel with seven types of loadable kernel modules.

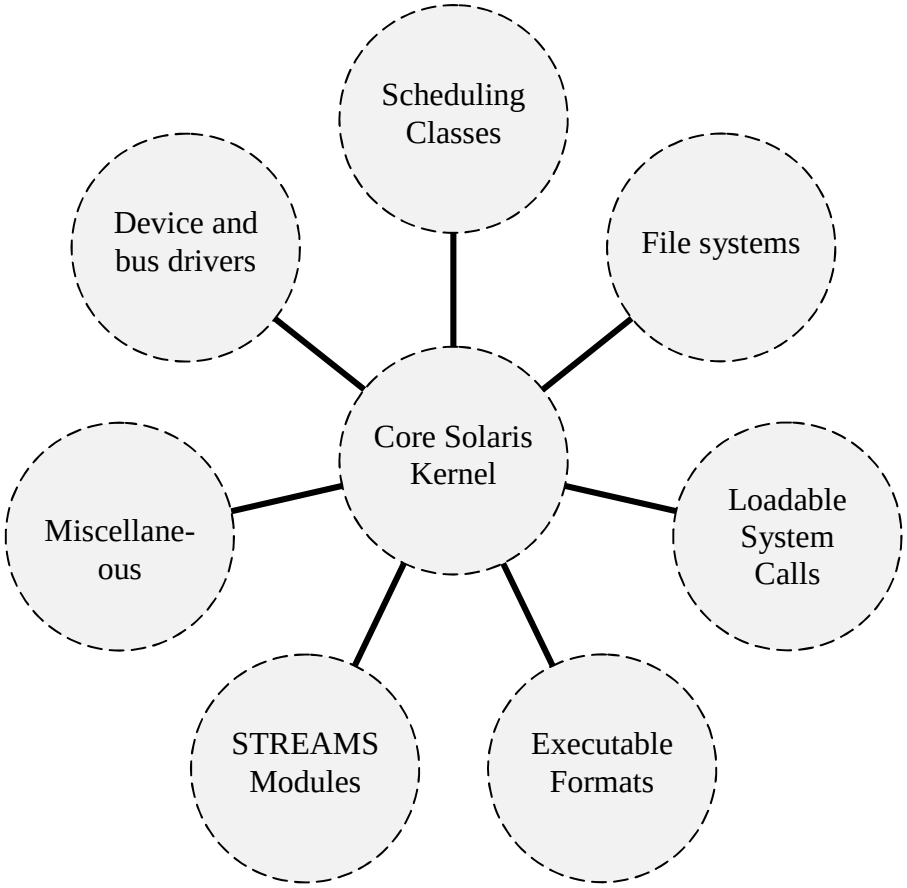


Fig 4.4 Solaris Loadable Modules

Such a design allows the kernel to provide core services yet also allows certain features to be implemented dynamically. For example, device and bus drivers for specific hardware can be added to the kernel, and support for different file systems can be added as loadable modules. The overall result resembles a layered system where each kernel section has defined, protected interfaces; but it is more flexible than a layered system where any module can call any other module.

Furthermore, the approach is like the microkernel approach where the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate. The Apple Mac OS X operating system uses a hybrid structure. It is a layered system in which one layer consists of the Mach microkernel.

The top layers include application environments and a set of services providing a graphical interface to applications. Below these

layers is the kernel environment, which consists primarily of the Mach microkernel and the BSD kernel. Mach provides memory management; support for remote procedure calls (RPCs) and inter-process communication (IPC) facilities, including message passing; and thread scheduling.

The BSD component provides a BSD command line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads. In addition to Mach and BSD, the kernel environment provides an i/o kit for development of device drivers and dynamically loadable modules (which Mac OS X refers to as kernel extensions). Applications and common services can make use of either the Mach or BSD facilities directly.

4.3 OPERATING SYSTEM GENERATION

Operating Systems may be designed and built for a specific hardware configuration at a specific site, but more commonly they are designed with a number of variable parameters and components, which are then configured for a particular operating environment.

Systems sometime need to be re-configured after the initial installation, to add additional resources, capabilities, or to tune performance, logging, or security.

Information that is needed to configure an OS include:

- What CPU(s) are installed on the system, and what optional characteristics does each have?
- How much RAM is installed? (This may be determined automatically, either at install or boot time.)
- What devices are present? The OS needs to determine which device drivers to include, as well as some device-specific characteristics and parameters.
- What OS options are desired, and what values to set for particular OS parameters. The latter may include the size of the open file table, the number of buffers to use, process scheduling (priority) parameters, disk scheduling algorithms, number of slots in the process table, etc.

At one extreme the OS source code can be edited, re-compiled, and linked into a new kernel.

More commonly configuration tables determine which modules to link into the new kernel, and what values to set for some key important parameters. This approach may require the configuration of complicated make files, which can be done either automatically or through interactive configuration programs; then

make is used to actually generate the new kernel specified by the new parameters.

At the other extreme a system configuration may be entirely defined by table data, in which case the "rebuilding" of the system merely requires editing data tables.

Once a system has been regenerated, it is usually required to reboot the system to activate the new kernel. Because there are possibilities for errors, most systems provide some mechanism for booting to older or alternate kernels.

4.4 SYSTEM BOOT

The general approach when most computers boot up goes something like this:

When the system powers up, an interrupt is generated which loads a memory address into the program counter, and the system begins executing instructions found at that address. This address points to the "bootstrap" program located in ROM chips (or EPROM chips) on the motherboard.

The ROM bootstrap program first runs hardware checks, determining what physical resources are present and doing power-on self tests (POST) of all HW for which this is applicable. Some devices, such as controller cards may have their own on-board diagnostics, which are called by the ROM bootstrap program.

The user generally has the option of pressing a special key during the POST process, which will launch the ROM BIOS configuration utility if pressed. This utility allows the user to specify and configure certain hardware parameters as where to look for an OS and whether or not to restrict access to the utility with a password.

Some hardware may also provide access to additional configuration setup programs, such as for a RAID disk controller or some special graphics or networking cards.

Assuming the utility has not been invoked, the bootstrap program then looks for a non-volatile storage device containing an OS. Depending on configuration, it may look for a floppy drive, CD ROM drive, or primary or secondary hard drives, in the order specified by the HW configuration utility.

Assuming it goes to a hard drive, it will find the first sector on the hard drive and load up the fdisk table, which contains information about how the physical hard drive is divided up into

logical partitions, where each partition starts and ends, and which partition is the "active" partition used for booting the system.

There is also a very small amount of system code in the portion of the first disk block not occupied by the fdisk table. This bootstrap code is the first step that is not built into the hardware, i.e. the first part which might be in any way OS-specific. Generally this code knows just enough to access the hard drive, and to load and execute a (slightly) larger boot program.

For a single-boot system, the boot program loaded off of the hard disk will then proceed to locate the kernel on the hard drive, load the kernel into memory, and then transfer control over to the kernel. There may be some opportunity to specify a particular kernel to be loaded at this stage, which may be useful if a new kernel has just been generated and doesn't work, or if the system has multiple kernels available with different configurations for different purposes. (Some systems may boot different configurations automatically, depending on what hardware has been found in earlier steps.)

For dual-boot or multiple-boot systems, the boot program will give the user an opportunity to specify a particular OS to load, with a default choice if the user does not pick a particular OS within a given time frame. The boot program then finds the boot loader for the chosen single-boot OS, and runs that program as described in the previous bullet point.

Once the kernel is running, it may give the user the opportunity to enter into single-user mode, also known as maintenance mode. This mode launches very few if any system services, and does not enable any logins other than the primary log in on the console. This mode is used primarily for system maintenance and diagnostics.

When the system enters full multi-user multi-tasking mode, it examines configuration files to determine which system services are to be started, and launches each of them in turn. It then spawns login programs (gettys) on each of the login devices which have been configured to enable user logins.

(The getty program initializes terminal I/O, issues the login prompt, accepts login names and passwords, and authenticates the user. If the user's password is authenticated, then the getty looks in system files to determine what shell is assigned to the user, and then "execs" (becomes) the user's shell. The shell program will look in system and user configuration files to initialize itself, and then issue prompts for user commands. Whenever the shell dies, either through logout or other means, then the system will issue a new getty for that terminal device.)

4.5 LET US SUM UP

- In MS-DOS, application programs are able to access the basic I/O routines to write directly to the display and disk drives.
- In layered approach, the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware, the highest (layer N) is the user interface.
- The main advantage of the layered approach is simplicity of construction and debugging
- In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called Mach that modularized the kernel using the microkernel approach
- The main function of the micro kernel is to provide a communication facility between the client program and the various services running in user space
- Tru64 UNIX (formerly Digital UNIX) provides a UNIX interface to the user, but it is implemented with a Mach kernel
- Most current UNIX-like systems, and Microsoft Windows, support loadable kernel modules, although they might use a different name for them, such as kernel loadable module (*kld*) in FreeBSD and kernel extension (*kext*) in OS X
- In addition to Mach and BSD, the kernel environment provides an i/o kit for development of device drivers and dynamically loadable modules (which Mac OS X refers to as kernel extensions).
- Once a system has been regenerated, it is usually required to reboot the system to activate the new kernel
- For dual-boot or multiple-boot systems, the boot program will give the user an opportunity to specify a particular OS to load, with a default choice if the user does not pick a particular OS within a given time frame.

4.6 UNIT END QUESTIONS

5. What are the differences between layered approach and microkernel approach?
6. What information is needed to configure an OS?
7. What do you mean by System Boot?
8. Define:
    a. Kernel loadable modules
    b. Maintenance mode

5

VIRTUAL MACHINES

Unit Structure

5.0 OBJECTIVES

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

- Study evolution of Virtual Machines.
- Distinguish between different Virtual Machines.

5.1 INTRODUCTION

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer. By using CPU scheduling and virtual-memory techniques, an operating system can create the illusion that a process has its own processor with its own (virtual) memory.

5.2 VIRTUAL MACHINES

The virtual machine provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer. Usually, the guest process is in fact an operating system, and that is how a single physical machine can run multiple operating systems concurrently, each in its own virtual machine.

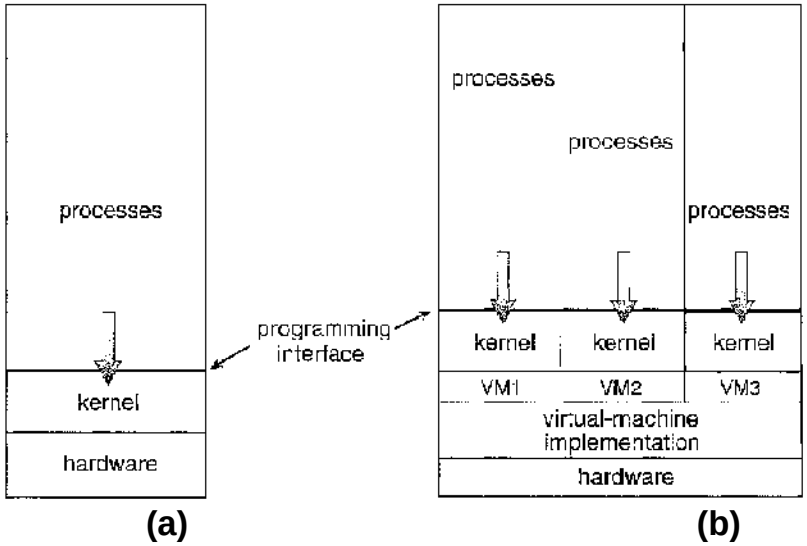


Fig 5.1 System models
(a) Non virtual machine (b) Virtual Machine

5.2.1 History

Virtual machines first appeared as the VM Operating System for IBM mainframes in 1972.

5.2.2 Benefits

Each OS runs independently of all the others, offering protection and security benefits. (Sharing of physical resources is not commonly implemented, but may be done as if the virtual machines were networked together.)

Virtual machines are a very useful tool for OS development, as they allow a user full access to and control over a virtual machine, without affecting other users operating the real machine.

As mentioned before, this approach can also be useful for product development and testing of SOFTWARE that must run on multiple Operating Systems / Hardware platforms.

5.2.3 Simulation

An alternative to creating an entire virtual machine is to simply run an emulator, which allows a program written for one OS to run

on a different OS. For example, a UNIX machine may run a DOS emulator in order to run DOS programs, or vice-versa. Emulators tend to run considerably slower than the native OS, and are also generally less than perfect.

5.2.4 Para-virtualization

Para-virtualization is another variation on the theme, in which an environment is provided for the guest program that is similar to its native OS, without trying to completely mimic it.

Guest programs must also be modified to run on the para-virtual OS.
Solaris 10 uses a zone system, in which the low-level hardware is not virtualized, but the OS and its devices (device drivers) are.

Within a zone, processes have the view of an isolated system, in which only the processes and resources within that zone are seen to exist. Figure 5.2 shows a Solaris system with the normal "global" operating space as well as two additional zones running on a small virtualization layer.

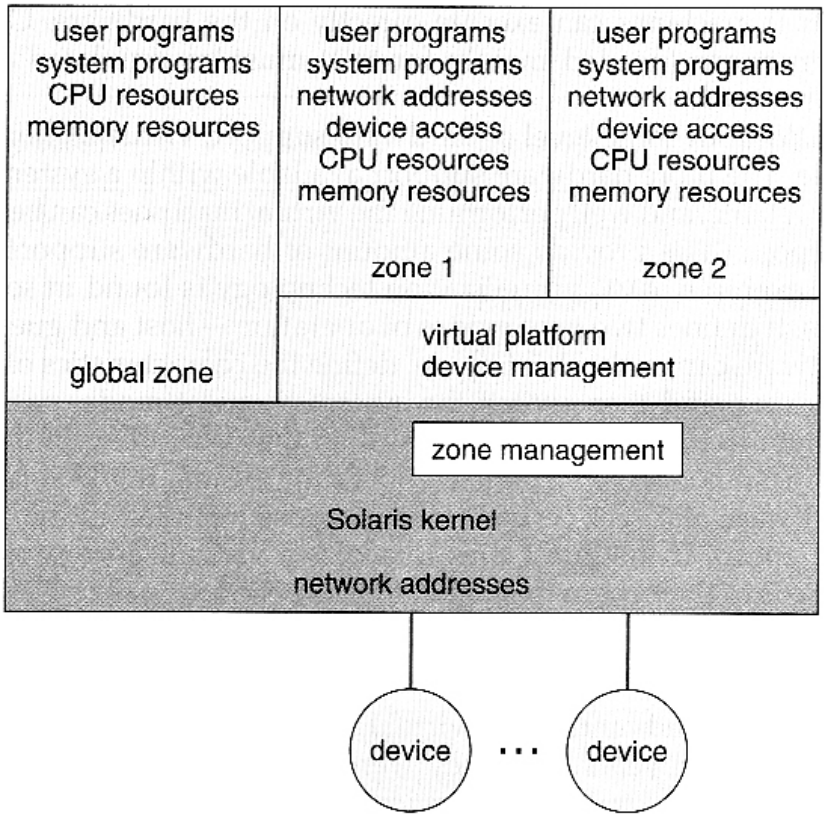


Fig 5.2 Solaris 10 with two containers

5.2.5 Implementation

Implementation may be challenging, partially due to the consequences of user versus kernel mode. Each of the

simultaneously running kernels needs to operate in kernel mode at some point, but the virtual machine actually runs in user mode.

So the kernel mode has to be simulated for each of the loaded Operating Systems, and kernel system calls passed through the virtual machine into a true kernel mode for eventual hardware access.

The virtual machines may run slower, due to the increased levels of code between applications and the hardware, or they may run faster, due to the benefits of caching. (And virtual devices may also be faster than real devices, such as RAM disks which are faster than physical disks).

5.2.6 Examples

5.2.6.1 VMware

VMware Workstation runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different guest operating systems as independent virtual machines. In this scenario, Linux is running as the host operating system; and FreeBSD, Windows NT, and Windows XP are running as guest operating systems. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems.

Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth. The physical disk the guest owns and manages is a file within the file system of the host operating system. To create an identical guest instance, we can simply copy the file. Copying the file to another location protects the guest instance against a disaster at the original site. Moving the file to another location moves the guest system. These scenarios show how virtualization can improve the efficiency of system administration as well as system resource use.

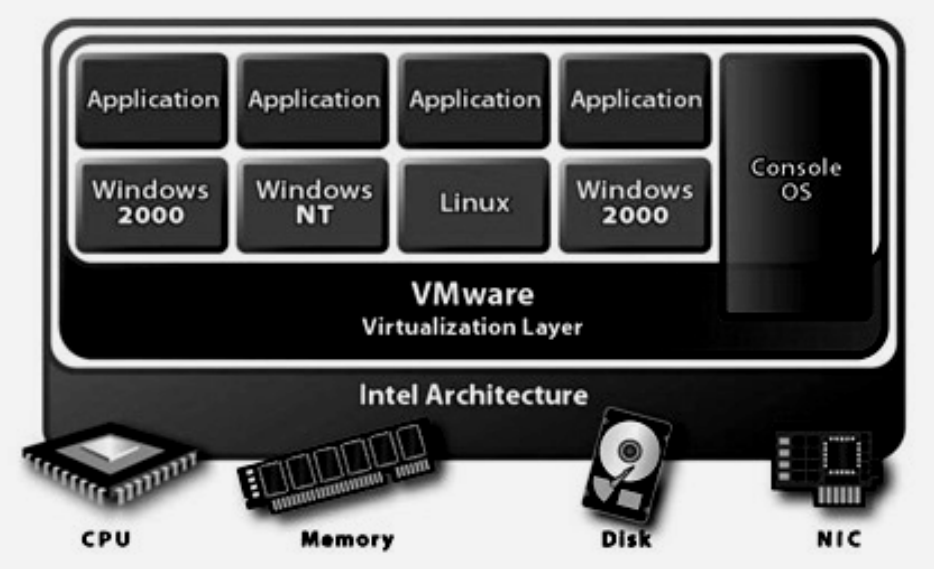


Fig 5.2 VMware Architecture

5.2.6.2 THE JAVA VIRTUAL MACHINE

Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library, Java also provides a specification for a Java virtual machine (JVM). Java objects are specified with the class construct; a Java program consists of one or more classes. For each Java class, the compiler produces an architecture-neutral bytecode output (.class) file that will run on any implementation of the JVM.

The JVM is a specification for an abstract computer. It consists of a class loader and a Java interpreter that executes the architecture-neutral bytecodes. The class loader loads the compiled class files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the class file is valid Java bytecode and does not overflow or underflow the stack. It also ensures the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter.

The JVM also automatically manages memory by performing garbage collection *(the practice of reclaiming memory from objects no longer in use and returning it to the system)*. Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.

The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part

of a Web browser. Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the bytecode operations one at a time.

A faster software technique is to use a just-in-time (JIT) compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system. These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions and the bytecode operations need not be interpreted all over again.

A technique that is potentially even faster is to run the JVM in hardware on a special Java chip that executes the Java bytecode operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

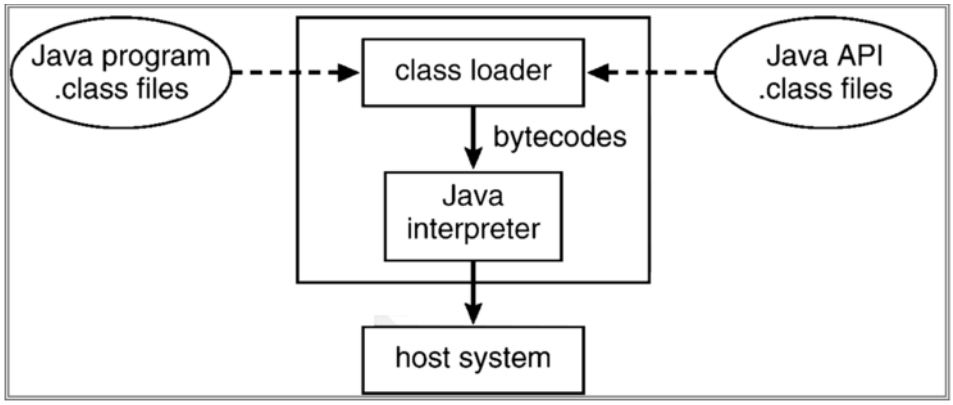


Fig 5.2 The JAVA Virtual Machine

5.2.6.2 THE .NET FRAMEWORK

The .NET Framework is a collection of technologies, including a set of class libraries, and an execution environment that come together to provide a platform for developing software. This platform allows programs to be written to target the .NET Framework instead of a specific architecture. A program written for the .NET Framework need not worry about the specifics of the hardware or the operating system on which it will run. Thus, any architecture implementing .NET will be able to successfully execute the program. This is because the execution environment abstracts these details and provides a virtual machine as an intermediary between the executing program and the underlying architecture.

At the core of the .NET Framework is the Common Language Runtime (CLR). The CLR is the implementation of the .NET virtual machine providing an environment for execution of programs written in any of the languages targeted at the .NET Framework.

Programs written in languages such as C# and VB.NET are compiled into an intermediate, architecture-independent language called Microsoft Intermediate Language (MS-IL). These compiled files, called assemblies, include MS-IL instructions and metadata. They have file extensions of either .EXE or .DLL. Upon execution of a program, the CLR loads assemblies into what is known as the Application Domain. As instructions are requested by the executing program, the CLR converts the MS-IL instructions inside the assemblies into native code that is specific to the underlying architecture using just-in-time compilation.

Once instructions have been converted to native code, they are kept and will continue to run as native code for the CPU. The architecture of the CLR for the .NET framework is shown in Figure 5.3.

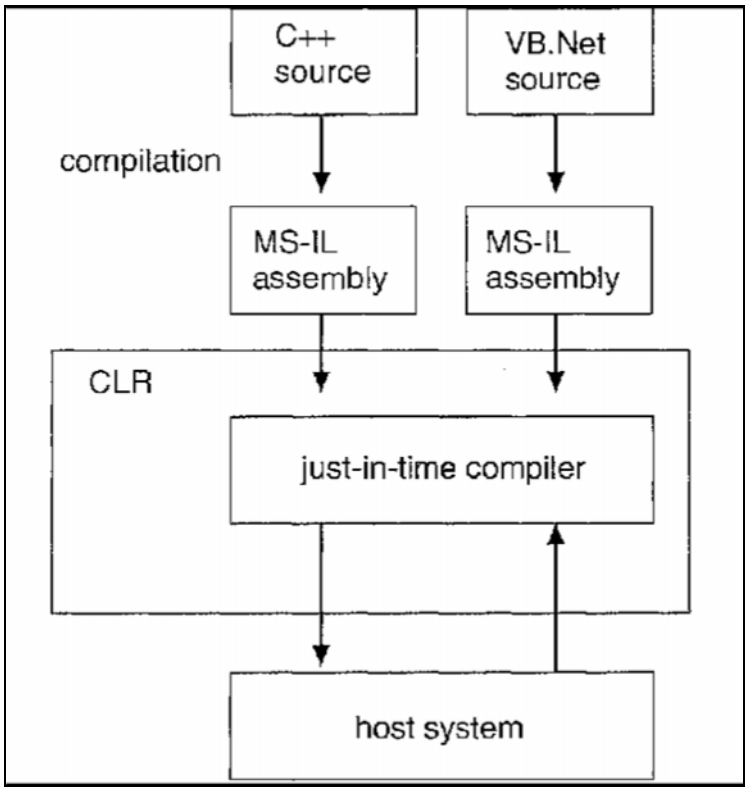


Fig 5.3 Architecture of the CLR for the .NET Framework

5.3 LET US SUM UP

- Virtual machines first appeared as the VM Operating System for IBM mainframes in 1972.

- Para-virtualization is another variation on the theme, in which an environment is provided for the guest program that is similar to its native OS, without trying to completely mimic it.

- Solaris 10 uses a zone system, in which the low-level hardware is not virtualized, but the OS and its devices (device drivers) are.
- The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems.

- It consists of a class loader and a Java interpreter that executes the architecture-neutral bytecodes.

- A faster software technique is to use a just-in-time (JIT) compiler. Here, the first time a Java method is invoked, the bytecodes for the method are turned into native machine language for the host system.

- The .NET Framework is a collection of technologies, including a set of class libraries, and an execution environment that come together to provide a platform for developing software.

5.4 UNIT END QUESTIONS

9. Write a short note on Virtual Machines.

10. Define : (a) Para-virtualization (b) JVM

11. Describe the architecture of :
    (a) The JAVA Virtual Machine
    (b) CLR for the .NET Framework

6

PROCESS

Unit Structure

6.0 OBJECTIVES

After reading this unit you will be able to:

- Define a process
- Study various process scheduling criteria

6.1 INTRODUCTION

The design of an operating system must be done in such a way that all requirement should be fulfilled.

- The operating system must interleave the execution of multiple processes, to maximize processor utilization while providing reasonable response time.

- The operating system must allocate resources to processes in conformance with a specific policy.

- The operating system may be required to support interprocess communication and user creation of processes.

6.2 PROCESS CONCEPTS

Process can be defined as:

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

A process is an entity that consists of a number of elements. Two essential elements of a process are **program code**, and a **set of data** associated with that code.

A process is more than the program code, which is sometimes known as the ***text section***. It also includes the current activity, as represented by the value of the ***program counter*** and the contents of the processor's registers. A process generally also includes the process ***stack***, which contains temporary data (such as function parameters, return addresses, and local variables), and a ***data section***, which contains global variables. A process may also include a ***heap***, which is memory that is dynamically allocated during process run time.

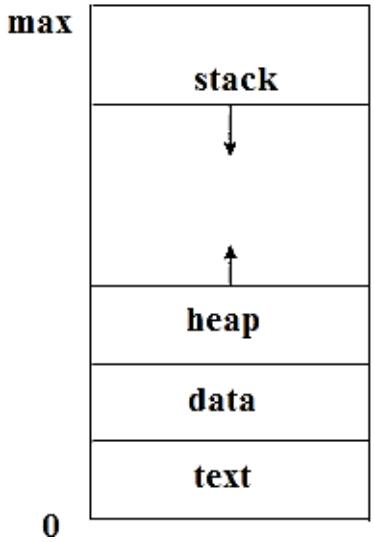


Fig 6.1 Process in Memory

6.2.1 PROCESS STATES

As a process executes, it changes state

- New: The process is being created
- Running: Instructions are being executed
- Waiting: The process is waiting for some event to occur
- Ready: The process is waiting to be assigned to a processor
- Terminated: The process has finished execution

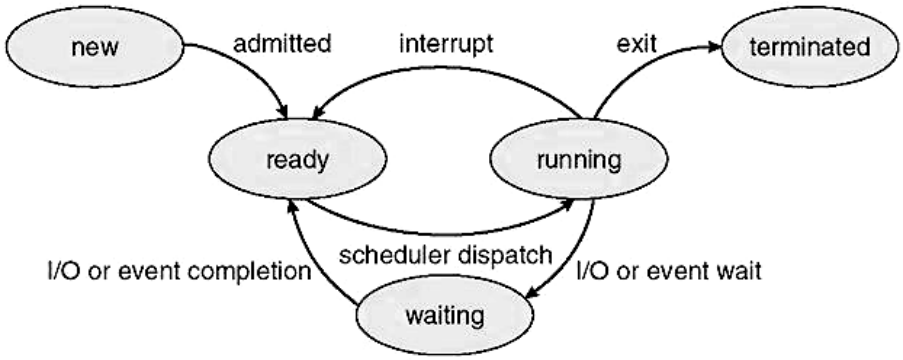


Fig 6.2 Block diagram of Process States

6.2.2 PROCESS CONTROL BLOCK

Identifier
State
Priority
Program Counter
Memory Pointers
Context data
I/O status information
Accounting information
• • •

Fig 6.3 Process Control Block (PCB)

Each process is described in the operating system by a process control block (PCB) also called a task control block. A PCB contains much of the information related to a specific process, including these:

Process state:
The state may be new, ready running, waiting, halted, and so on.

Program counter:
The counter indicates the address of the next instruction to be executed for this process.

CPU registers:
The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

CPU-scheduling information:
This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory-management information:
This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

Accounting information:
This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

I/O status information:
This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

6.2.3 THREADS

A single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

On a system that supports threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

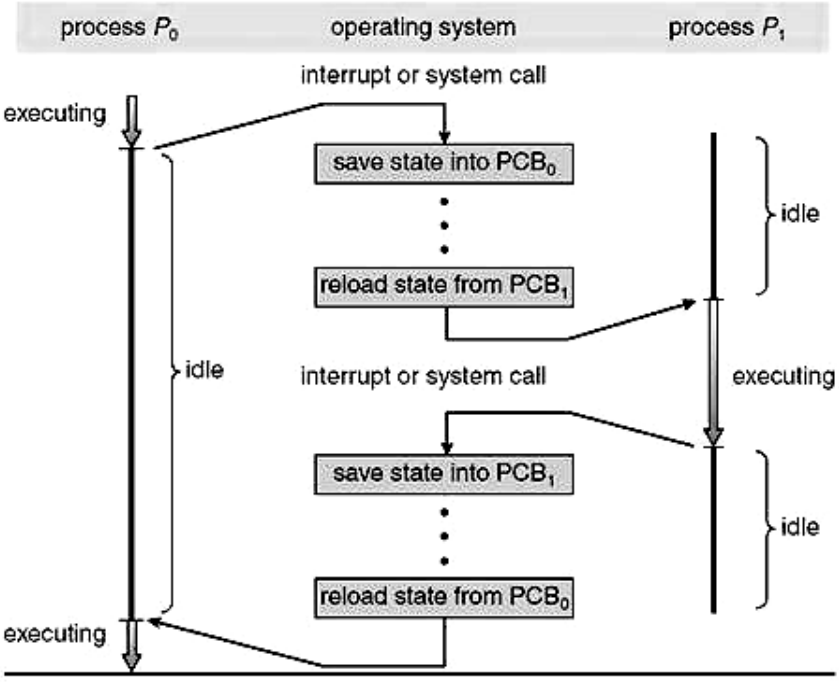


Fig 6.4 CPU switch from process to process

6.3 PROCESS SCHEDULING

When a computer is *multiprogrammed*, it frequently has multiple processes or threads competing for the CPU at the same time. This situation occurs whenever two or more of them are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the ***scheduler***, and the algorithm it uses is called the ***scheduling algorithm***.

6.4 SCHEDULING CRITERIA

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Some goals depend on the environment (batch, interactive, or real time), but there are also some that are desirable in all cases. Some goals (scheduling criteria) are listed below.

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

6.5 LET US SUM UP

Process can be defined as:

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

A process generally also includes the process ***stack***, which contains temporary data (such as function parameters, return addresses, and local variables), and a ***data section***, which contains global variables.

As a process executes, it changes state

- New: The process is being created
- Running: Instructions are being executed
- Waiting: The process is waiting for some event to occur
- Ready: The process is waiting to be assigned to a processor
- Terminated: The process has finished execution

The part of the operating system that makes the choice is called the ***scheduler***, and the algorithm it uses is called the ***scheduling algorithm***.

6.6 UNIT END QUESTIONS

1. Define a process.
2. Describe various scheduling criteria.
3. What are threads?
4. What are the components of a Process Control Block?

7

PROCESS SCHEDULING ALGORITHMS

Unit Structure

7.0 OBJECTIVES

After reading this unit you will be able to:
- Discuss various scheduling algorithms

7.1 INTRODUCTION

Scheduling algorithms in Modern Operating Systems are used to:
- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time

7.2 SCHEDULING ALGORITHMS

7.2.1 FIRST-COME, FIRST-SERVED (FCFS) SCHEDULING

Requests are scheduled in the order in which they arrive in the system. The list of pending requests is organized as a queue. The scheduler always schedules the first request in the list. An

example of FCFS scheduling is a batch processing system in which jobs are ordered according to their arrival times (or arbitrarily, if they arrive at exactly the same time) and results of a job are released to the user immediately on completion of the job. The following example illustrates operation of an FCFS scheduler.

Table 7.1 Processes for scheduling

Process	P_1	P_2	P_3	P_4	P_5
Admission time	0	2	3	4	8
Service time	3	3	5	2	3

Time	**Completed Process**			**Processes in system (in FCFS order)**	**Scheduled process**
	id	***ta***	***w***		
0	-	-	-	P_1	P_1
3	P_1	3	1.00	P_2, P_3	P_2
6	P_2	4	1.33	P_3, P_4	P_3
11	P_3	8	1.60	P_4, P_5	P_4
13	P_4	9	4.50	P_5	P_5
16	P_5	8	2.67	-	-

ta = 7.40 seconds
w = 2.22

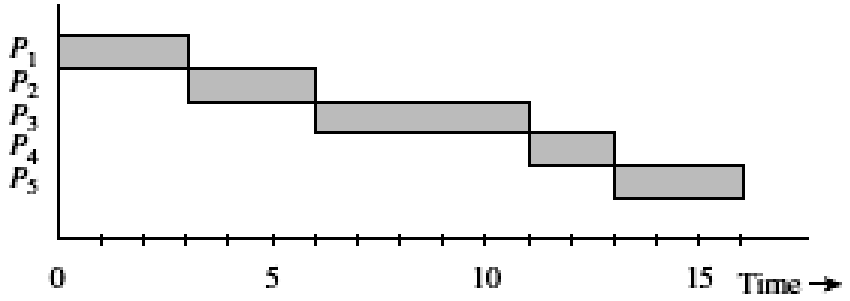


Fig 7.1 Scheduling using FCFS policy

Figure 7.1 illustrates the scheduling decisions made by the FCFS scheduling policy for the processes of Table 7.1. Process P1 is scheduled at time 0. The pending list contains P2 and P3 when P1 completes at 3 seconds, so P2 is scheduled. The Completed column shows the id of the completed process and its turnaround time (ta) and weighted turnaround (w). The mean values of ta and w (i.e., ta and w) are shown below the table. The timing chart of Figure 7.1 shows how the processes operated.

From the above example, it is seen that considerable variation exists in the weighted turnarounds provided by FCFS scheduling. This variation would have been larger if processes subject to large

turnaround times were short -e.g., the weighted turnaround of P4 would have been larger if its execution requirement had been 1 second or 0.5 second.

7.2.2 SHORTEST JOB FIRST SCHEDULING

Shortest Job First scheduling assumes the run times are known in advance. In an insurance company, for example, people can predict quite accurately how long it will take to run a batch of 1000 claims, since similar work is done every day. When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest job first. Look at Fig. 7.7. Here we find four jobs A, B, C, and D with run times of 8, 4, 4, and 4 minutes, respectively. By running them in that order, the turnaround time for A is 8 minutes, for B is 12 minutes, for C is 16 minutes, and for D is 20 minutes for an average of 14 minutes.

(a) (b)

Fig 7.2 (a) Running four jobs in the original order
(b) Running them in shortest job first order

Now let us consider running these four jobs using shortest job first, as shown in Fig. 7.2 (b). The turnaround times are now 4, 8, 12, and 20 minutes for an average of 11 minutes. Shortest job first is probably optimal. Consider the case of four jobs, with run times of A, B, C, and D, respectively. The first job finishes at time a, the second finishes at time a + b, and so on. The mean turnaround time is (4a + 3b + 2c + d)/4. It is clear that A contributes more to the average than the other times, so it should be the shortest job, with b next, then C, and finally D as the longest as it affects only its own turnaround time. The same argument applies equally well to any number of jobs.

It is worth pointing out that shortest job first is only optimal when all the jobs are available simultaneously. As a counterexample, consider five jobs, A through E, with run times of 2, 4, 1, 1, and 1, respectively. Their arrival times are 0, 0, 3, 3, and 3. Initially, only A or B can be chosen, since the other three jobs have not arrived yet. Using shortest job first we will run the jobs in the order A, B, C, D, E, for an average wait of 4.7. However, running them in the order B, C, D, E, A has an average wait of 4.4.

7.2.3 PRIORITY SCHEDULING

The basic idea is straightforward: each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are

scheduled in FCFS order. The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm.

A SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. That is, the longer the CPU burst, the lower the priority and vice versa.

Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities or qualities to compute priority of a process.

Examples of Internal priorities are

- Time limits.
- Memory requirements.
- File requirements, for example, number of open files.
- CPU v/s I/O requirements.

Externally defined priorities are set by criteria that are external to operating system such as

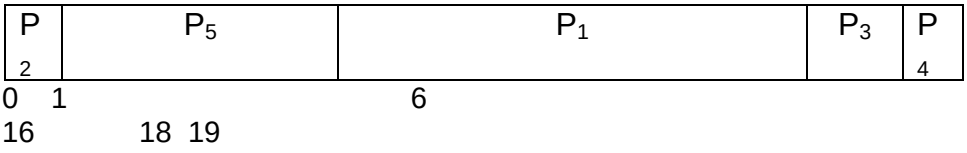
- The importance of process.
- Type or amount of funds being paid for computer use.
- The department sponsoring the work.
- Politics.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P_1, P_2, · · ·, P_5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Table 7.2 Processes for Priority scheduling

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.
Priority scheduling can be either preemptive or non-preemptive

- A preemptive priority algorithm will preemptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
- A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling is indefinite *blocking* or *starvation*. A solution to the problem of indefinite blockage of the low-priority process is *aging*. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

7.2.4 ROUND ROBIN SCHEDULING

One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR). In the round robin scheduling, processes are dispatched in a FIFO (First-In-First-Out) manner but are given a limited amount of CPU time called a ***time-slice*** or a ***quantum***.

If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.

Figure 7.3 summarizes operation of the RR scheduler with δ = 1 second for the five processes shown in Table 7.3. The scheduler makes scheduling decisions every second. The time when a decision is made is shown in the first row of the table in the top half of Figure 7.3. The next five rows show positions of the five processes in the ready queue. A blank entry indicates that the process is not in the system at the designated time. The last row shows the process selected by the scheduler; it is the process occupying the first position in the ready queue.

Time of scheduling		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Position of	P_1	1	1	2	1												
Processes in	P_2			1	3	2	1	3	2	1							
***ready* queue**	P_3				2	1	3	2	1	4	3	2	1	2	1	2	1
(1 implies	P_4					3	2	1	3	2	1						
head of queue)	P_5									3	2	1	2	1	2	1	
Process scheduled		P_1	P_1	P_2	P_1	P_3	P_2	P_4	P_3	P_2	P_4	P_5	P_3	P_5	P_3	P_5	P_3

c	t_a	w
4	4	1.33
9	7	2.33
16	13	2.60
10	6	3.00
15	7	2.33

Table 7.3 Processes for RR scheduling

t_a = 7.4 seconds, w = 2.32,
c: completion time of a process

Consider the situation at 2 seconds. The scheduling queue contains P2 followed by P1. Hence P2 is scheduled. Process P3 arrives at 3 seconds, and is entered in the queue. P2 is also preempted at 3 seconds and it is entered in the queue. Hence the queue has process P1 followed by P3 and P2, so P1 is scheduled.

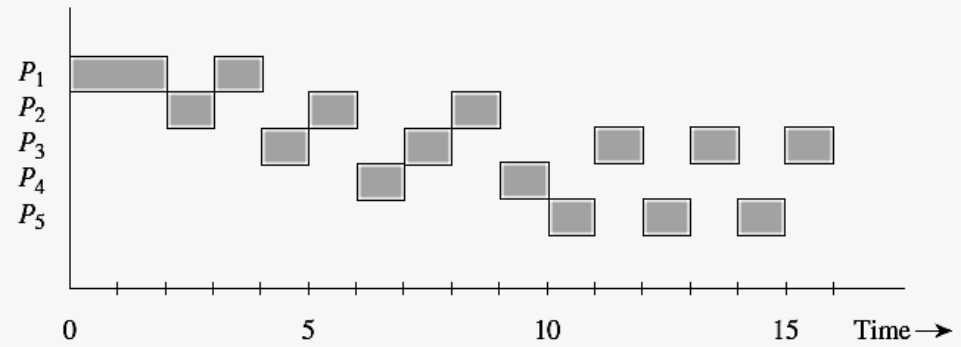


Fig 7.3 Scheduling using Round-robin policy with time-slicing

The turnaround times and weighted turnarounds of the processes are as shown in the right part of the table. The c column shows completion times. The turnaround times and weighted turnarounds are inferior to those given by the non-preemptive policies because the CPU time is shared among many processes because of time-slicing.

It can be seen that processes P_2, P_3, and P_4, which arrive at around the same time, receive approximately equal weighted turnarounds. P_4 receives the worst weighted turnaround because through most of its life it is one of three processes present in the system. P_1 receives the best weighted turnaround because no other process exists in the system during the early part of its execution. Thus weighted turnarounds depend on the load in the system.

Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.

The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and appoximates FCFS.

In any event, the average waiting time under round robin scheduling is often quite long.

7.2.5 MULTILEVEL QUEUE SCHEDULING

A multilevel queue scheduling algorithm partitions the **Ready queue** is partitioned into separate queues:

- **foreground** (interactive)
- **background** (batch)

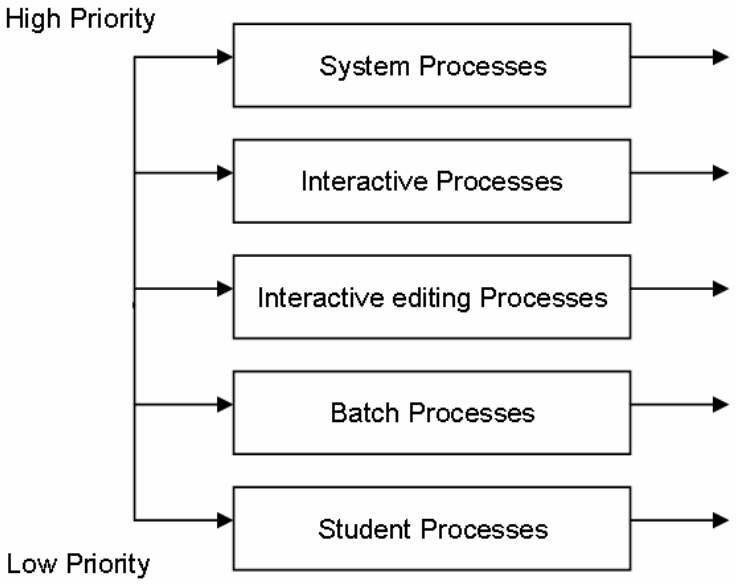


Fig 7.4 Multilevel queue scheduling

In a multilevel queue scheduling processes are permanently assigned to one queues. The processes are permanently assigned to one another, based on some property of the process, such as

- Memory size
- Process priority
- Process type

Algorithm choose the process from the occupied queue that has the highest priority, and run that process either

- Preemptive or
- Non-preemptive

Each queue has its own scheduling algorithm

- foreground – RR
- background – FCFS

Possibility I

If each queue has absolute priority over lower-priority queues then no process in the queue could run unless the queue for the highest-priority processes were all empty.

For example, in the above figure no process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes will all empty.

Possibility II

If there is a time slice between the queues then each queue gets a certain amount of CPU times, which it can then schedule among the processes in its queue. For instance;

- 80% of the CPU time to foreground queue using RR.
- 20% of the CPU time to background queue using FCFS.

Since processes do not move between queues so, this policy has the advantage of low scheduling overhead, but it is inflexible.

7.2.6 MULTILEVEL FEEDBACK QUEUE SCHEDULING

Here, processes are not permanently assigned to a queue on entry to the system. Instead, they are allowed to move between queues. The idea is to separate processes with different CPU burst characteristics. If a process uses too much CPU time, it will be moved to a lower priority queue. Similarly, a process that waits too long in a low priority queue will be moved to a higher priority queue. This form of aging prevents starvation.

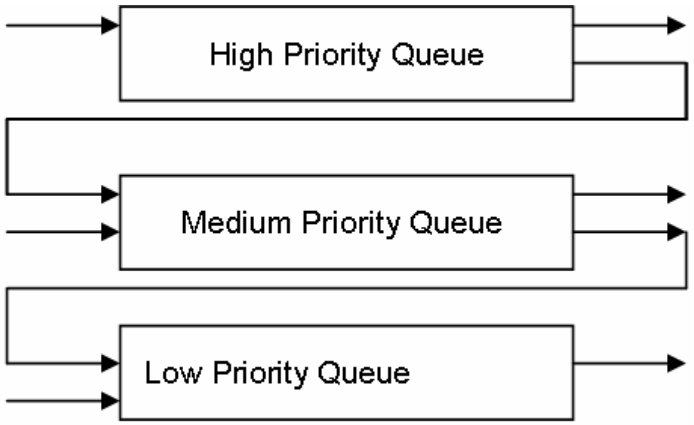


Fig 7.5 Multilevel Feedback Queue Scheduling

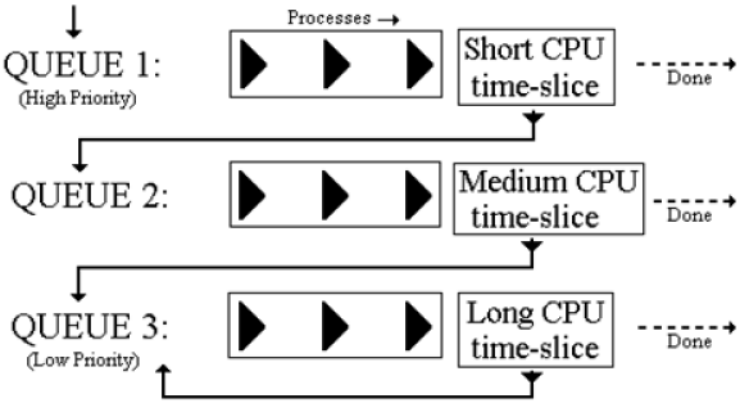


Fig 7.6 MFQ Scheduling architecture

Multilevel feedback queue scheduler is characterized by the following parameters:

1. Number of queues
2. Scheduling g algorithms for each queue
3. Method used to determine when to upgrade a process
4. Method used to determine when to demote a process
5. Method used to determine which queue a process will enter when that process needs service

Example:
Three queues:

1. Q0 – time quantum 8 milliseconds
2. Q1 – time quantum 16 milliseconds
3. Q2 – FCFS

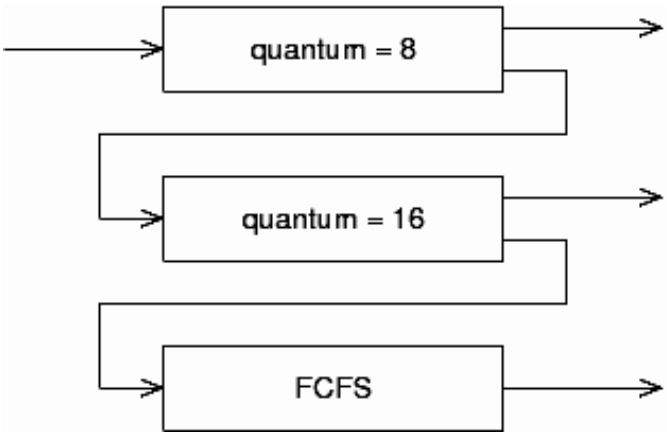


Fig 7.8 MFQ scheduling example

Scheduling:

1. A new job enters queue Q0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q1.
2. At Q1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q2.

7.3 LET US SUM UP

- The scheduler always schedules the first request in the list.
- Shortest Job First scheduling assumes the run times are known in advance.
- It is worth pointing out that shortest job first is only optimal when all the jobs are available simultaneously.
- The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm.
- Priority can be defined either internally or externally.

- A major problem with priority scheduling is indefinite *blocking* or *starvation*.
- In the round robin scheduling, processes are dispatched in a FIFO (First-In-First-Out) manner but are given a limited amount of CPU time called a ***time-slice*** or a ***quantum***.
- Round Robin Scheduling is preemptive (at the end of time-slice).
- A multilevel queue scheduling algorithm partitions the **Ready queue** is partitioned into separate queues
- In MFQ scheduling, processes are not permanently assigned to a queue on entry to the system.

7.4 UNIT END QUESTIONS

1. Define
    a. Quantum
    b. Aging
2. Give an example of First-Come, First-Served Scheduling.
3. What is the difference between Multilevel Queue Scheduling and Multilevel Feedback Queue Scheduling?
4. Describe the architecture of MFQ scheduling with the help of diagrams.
5. State the criteria for internal and external priorities.

8

PROCESS AND THREAD MANAGEMENT

Unit Structure

8.0 OBJECTIVES

After reading this unit you will be able to:

- Describe the creation & termination of a process
- Study various interprocess communications
- Introduce the notion of a thread- a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- Examine issues related to multithreaded programming.

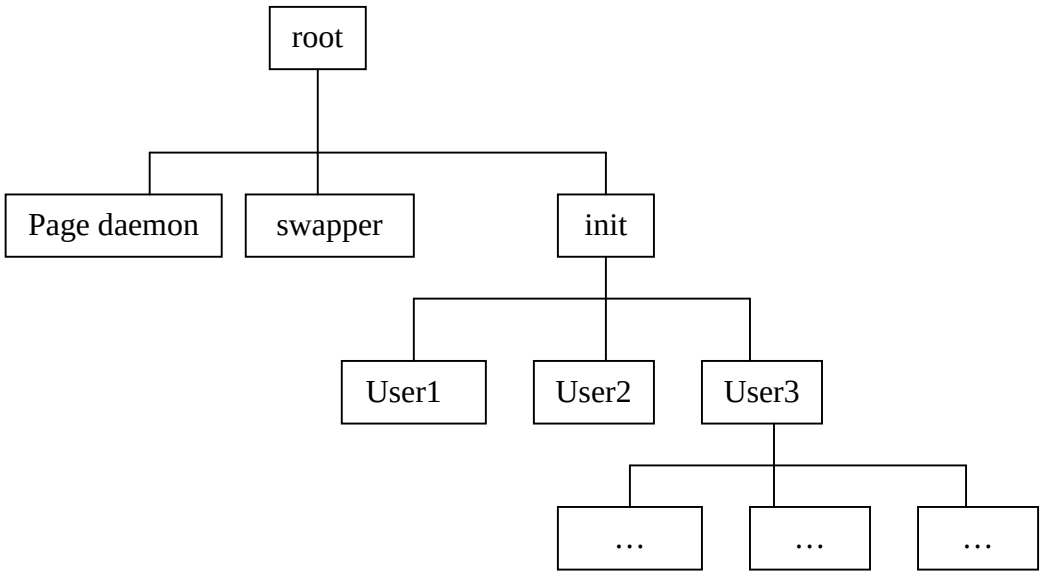
8.1 INTRODUCTION

Generally, when a thread finishes performing a task, thread is suspended or destroyed. Writing a program where a process creates multiple threads is called multithread programming. It is the ability by which an OS is able to run different parts of the same program simultaneously. It offers better utilization of processors and other system resources. For example, word processor makes use of multi-threading – in the foreground it can check spelling as well as save document in the background.

8.2 OPERATIONS ON PROCESS

8.2.1 PROCESS CREATION

A process may create several new processes, via a ***create-process system call***, during the course of execution. The creating process is called a ***parent process***, whereas the new processes are called the ***children*** of that process. Each of these new processes may in turn create other processes, forming a *tree* of processes as shown below.



A tree of processes on a typical UNIX system

Fig 8.1

In general, a process will need certain resources (such as CPU time, memory, files, I/O devices) to accomplish its task. When the process creates a sub process, that sub process may be able to obtain its resources directly from the OS, or it may be controlled to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevent any process from overloading the system by creating too many sub processes.

When a process is created it obtains along with the resources, initialization data (or input from the file, say F1) that may be passed along from the parent process to the child process. It may also get the name of the output device. New process may get two open files, F1 and the terminal device, and may just need to transfer the datum between the two.

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

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

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.

2. The child process has a program loaded into it.

Following are some reasons for creation of a process:

- User logs on.
- User starts a program.
- Operating systems creates process to provide service, e.g., to manage printer.
- Some program starts another process, e.g., Netscape calls *xv* to display a picture.

In UNIX, each process is identified by its **process identifier (PID)**, which is a unique integer. A new process is created by the *fork* system call. The new process consists of a copy of the address space of the original process which helps the parent process to communicate easily with its child process. Both processes (the parent & the child) continue execution at the instruction after the fork system call, with one difference: The return code for the fork system call is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

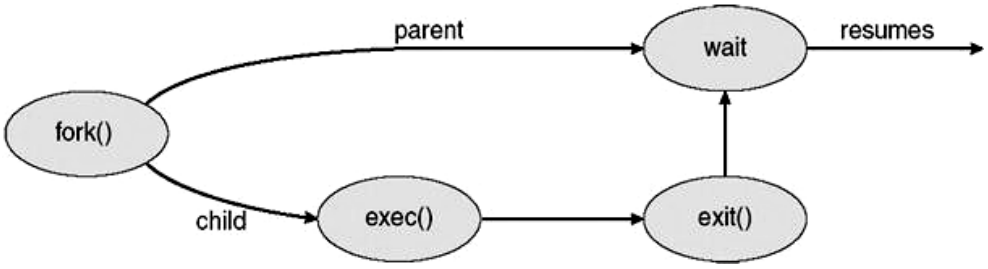


Fig 8.2 UNIX Process Creation

To understand the above possibilities, consider the following C program:

```
int main() {
            Pid_t  pid;
           /* fork another process */
```

```
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

The parent creates a child process using the ***fork*** system call. We now have two different processes running a copy of the same program. The value of the pid for the child process is zero; that for the parent is an integer value greater than zero. The child process overlays its address space with the UNIX command ***/bin/ls*** (used to get a directory listing) using the ***execlp*** system call. The parent waits for the child process to complete with the ***wait*** system call. When the child process completes, the parent process resumes the call to wait where it completes using the ***exit*** system call.

8.2.2 PROCESS TERMINATION

A process terminates when it finishes executing its final statement and asks the OS to delete it by using the ***exit*** system call. At that point, the process may return data (output) to its parent process (via the ***wait*** system all). All the resources of the process, including physical and virtual memory, open files, and I/O buffers are deallocated by the OS.

A process can cause the termination of another process via an appropriate system call such as ***abort.*** Usually, only the parent of the process that is to be terminated can invoke such a system call otherwise you can arbitrarily kill each other's jobs.

A parent may terminate the execution of one of its children for the following reasons:

- The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
- The task assigned to the child is no longer required.

- The parent is exiting, and the OS does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This is referred to as **Cascading Termination**, and is normally initiated by the OS. In the case of UNIX, if the parent terminates, however, all its children have assigned as the new parent the ***init*** process. Thus, the children still have a parent to collect their status and execution statistics.

The new process terminates the existing process, usually due to following reasons:

- **Normal Exit**
  Most processes terminates because they have done their job. This call is exist in UNIX.

- **Error Exit**
  When process discovers a fatal error. For example, a user tries to compile a program that does not exist.

- **Fatal Error**
  An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.

- **Killed by another Process**
  A process executes a system call telling the Operating Systems to terminate some other process. In UNIX, this call is kill. In some systems when a process kills all processes it created are killed as well (UNIX does not work this way).

8.2.2 COOPERATING PROCESSES

A process is **independent** if it cannot affect or be affected by the other processes executing in the system. On the other hand, a process is **cooperating** if it can affect or be affected by the other processes executing in the system. Process cooperation is required for the following reasons:

- **Information sharing:**
  Several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to these types of resources.

- **Computation speedup:**
  If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the

computer has multiple processing elements (such as CPUs or I/O channels).

- **Modularity:**
  To construct the system in a modular fashion, dividing the system functions into separate processes or threads.

- **Convenience:**
  Individual user may have many tasks to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

Concurrent execution of cooperating processes requires mechanism that allow processes to communicate with one another and to synchronize their actions.

8.3 INTERPROCESS COMMUNICATION

The OS provides the means for cooperating processes to communicate with each other via an interprocess communication (IPC) facility. IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. IPC is particularly useful in a distributed environment where the communicating processes may reside on different computers connected with a network e.g. **chat** program used on the **world wide web.** IPC is best provided by a **message-passing system**, and the message systems can be defined in many ways.

8.3.1 MESSAGE PASSING SYSTEM

Message system allows processes to communicate with one another without the need to resort to shared data. Services are provided as ordinary user processes operate outside the kernel. Communication among the user processes is accomplished through the passing of messages. An IPC facility provides at least two operations: send (message) and receive (message). Messages sent by a process can be of either fixed or variable size.

If processes **P** and **Q** want to communicate, they must **send** messages to **send** and **receive** from each other; a **communication link** must exist between them. There are several methods for logical implementation of a link as follows:

- Direct or indirect communication.
- Symmetric or asymmetric communication.
- Automatic or explicit buffering.
- Send by copy or send by reference.
- Fixed-sized or variable-sized message.

8.3.1.1 DIRECT COMMUNICATION

Each process that wants to communicate must explicitly **name** the **recipient** or **sender** of the communication. The send and receive primitives are defied as:

- **send (P, message)** – Send a message to process **P**.
- **receive (Q, message)**–Receive a message from process **Q**.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

This scheme exhibits **symmetry in addressing**; that is, both the sender and the receiver processes must name the other to communicate.

A variant of this scheme employs **asymmetry in addressing**. Only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send and receive primitives are as follows:

- **send (P, message)** – Send a message to process **P**.
- **receive (id, message)** – Receive a message from any process; the variable **id** is set to the name of the process with which communication has taken place.

The disadvantage in both schemes:

Changing the name of a process may necessitate examining all other process definitions. All references to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the viewpoint of separate compilation.

8.3.1.2 INDIRECT COMMUNICATION

The messages are sent to and received from **mailboxes**, or **ports**. Each mailbox has a unique identification. Two processes can communicate only if they share a mailbox. The send and receive primitives are defined as follows:

- **send (A, message) -** Send a message to mailbox **A**.
- **receive (A,message) –** Receive a message from mailbox **A**.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.

- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

If processes P_1, P_2 and P_3 all share mailbox A. Process P_1 sends a message to A, while P_2 and P_3 each execute and receive from A. The process to receive the message depends on one of the scheme that:

- Allows a link to be associated with at most two processes.
- Allows utmost one process at a time to execute a receive operation.
- Allows the system to select arbitrarily which process will receive the message (that is either P_2 or P_3, but not both, will receive the message). The system may identify the receiver to the sender.

If the mailbox is owned by process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox).

When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists. On the other hand, a mailbox owned by the OS is independent and is not attached to any particular process. The OS then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.
- Process who create a mailbox is the owner by default and receives messages through this mail box. Ownership can be changed by OS through appropriate system calls to provide multiple receivers for each mailbox.

8.3.1.3 SYNCHRONIZATION

The **send** and **receive** system calls are used to communicate between processes but there are different design options for implementing these calls. Message passing may be either **blocking** or **non-blocking -** also known as **synchronous** and **asynchronous.**

- **Blocking send:**
  The sending process is blocked until the message is received by the receiving process or by the mailbox.

- **Non-blocking send:**

The sending process sends the message and resumes operation.

- **Blocking receive:**
  The receiver blocks until a message is available.

- **Non-blocking receive**:
  The receiver retrieves either a valid message or a null.

Different combinations of send and receive are possible. When both the send and receive are blocking, we have a **rendezvous** (to meet) between the sender and receiver.

8.3.1.4 BUFFERING

During direct or indirect communication, messages exchanged between communicating processes reside in a temporary queue which are implemented in the following three ways:

- **Zero capacity:**

  The queue has maximum length 0; thus, the link cannot have any message waiting in it. In this case, the sender must block until the recipient receives the message. This is referred to as no buffering.

- **Bounded capacity:**

  The queue has finite length *n*; thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. If the link is full, the sender must block until space is available in the queue. This is referred to as auto buffering

- **Unbounded capacity:**

  The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks. This also referred to as auto buffering.

8.4 MULTITHREADING MODELS

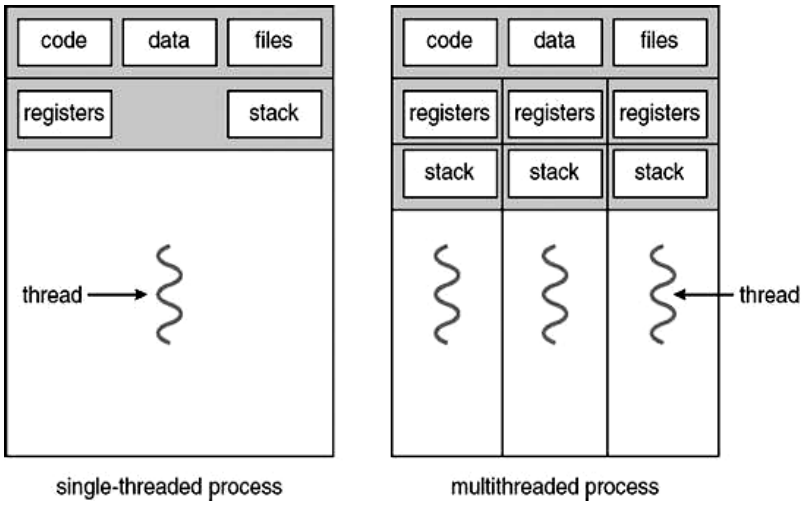


Fig 8.3 Single threaded and multithreaded processes

Support for threads may be provided either at the user level, for or by the kernel, for threads. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. A relationship exists between user threads and kernel threads.

There are mainly three types of multithreading models available for user and kernel threads.

8.4.1 Many-to-One Model:

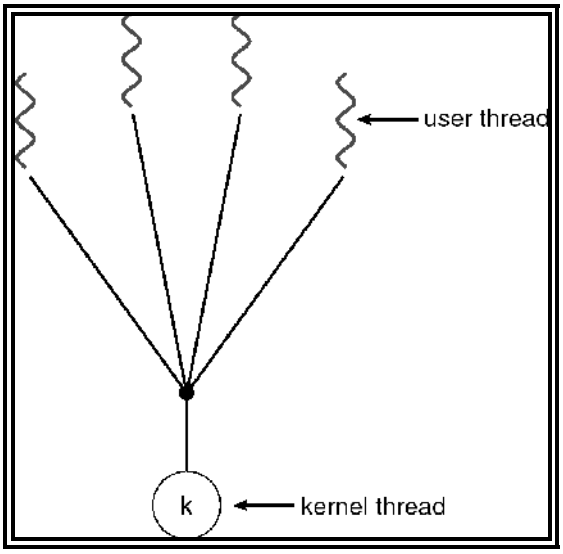


Fig 8.4 Many-to-One model

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done in user space, so it is efficient, but the entire process will block if a thread makes a blocking system call. Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. Green threads- a thread library available for Solaris uses this model. OS that do not support the kernel threads use this model for user level threads.

8.4.2 One-to-One Model:

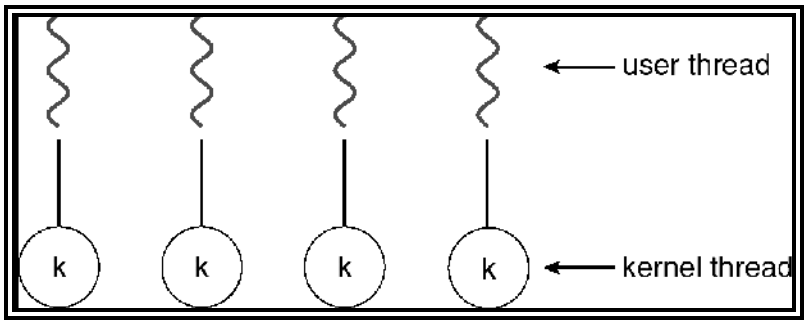


Fig 8.5 One-to-One model

Maps each user thread to a kernel thread. Allows another thread to run when a thread makes a blocking system call. Also allows multiple threads to run in parallel on multiprocessors. Drawback is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, and therefore, restriction has to be made on creation of number of threads. Window NT, Windows 2000, and OS/2 implement this model.

8.4.3 Many-to-Many Model:

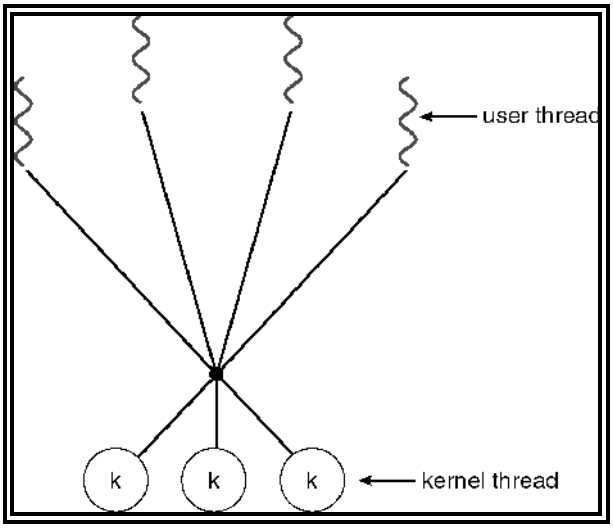


Fig 8.6 Many-to-Many model

Multiplexes many user-level threads to smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor). Model allows the developer to create as many user threads as he wishes, true concurrency is not gained because the kernel can schedule only one thread at a time, but the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution. Solaris 2, IRIX, HP-UX, and TRU64 UNIX support this model.

8.5 THREADING ISSUES

Generally six issues are considered for threading:

1. Semantics of **fork()** and **exec()** system calls
2. Thread cancellation
3. Signal handling
4. Thread pools
5. Thread specific data
6. Scheduler activations

8.5.1 The fork and exec System Calls:

Fork system call is used to create a separate, duplicate process. In a multithreaded program, the semantics of the fork and exec system calls change. If one thread in a program calls fork, does the new process duplicate all threads or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork, one that duplicates all threads and another that duplicates only the thread that invoked the fork system call. If a

thread invokes the exec system call, the program specified in the parameter to exec will replace the entire process-including all threads and LWPs.

8.5.2 Thread cancellation:

It is the task of terminating thread before it has completed. E.g., if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled. A thread that is to be cancelled is often referred to as the **target thread.**

Cancellation of threads may be one of the following types:

1) **Asynchronous cancellation**: One thread immediately terminates the target thread. The OS will often reclaim system resources from a cancellation thread, but often will not reclaim all resources.

2) **Deferred cancellation**: The target thread can periodically check if it should terminate, allowing the target thread an opportunity to terminate itself in an orderly fashion. This allows a thread to check if it should be canceled at a point when it can safely be cancelled. Pthread refers to such points as **cancellation points.**

8.5.3 Signal Handling:

A signal is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously depending upon the source and the reason for the event being signaled.

All signals follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

Every signal may be handled by one of two possible handlers:

1. A default signal handler that is run by the kernel.
2. A user-defined signal handler.

The following options exist in delivering the signals in a multithreaded programs:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process
4. Assign a specific thread to receive all signals for the process.

8.5.4 Thread pools:

The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool- if one is available-passing it the request to service. Once the thread completes its service, it returns to the pool awaiting more work. If the pool contains no available thread, the server waits until one becomes free. The number of threads in the pool depends upon the number of CPUs, the amount of physical memory, and the expected number of concurrent client requests.

In particular, the benefits of thread pools are:

1. It is usually faster to service a request with an existing thread than waiting to create a thread.

2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

8.5.5 Thread specific data:

Threads belonging to a process share the data of the process. This sharing of data provides one of the benefits of multithreaded programming. However, each thread might need its own copy of certain data in some circumstances, called as **thread-specific data.** Most thread libraries- including Win32 and Pthreads-provide some form of support for thread specific data.

8.5.6 Scheduler Activations:

Communication between the kernel and the thread library may be required by the many-to-many and two-level models. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance. Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure-typically known as a lightweight process, or LWP-is shown in Figure 8.7.

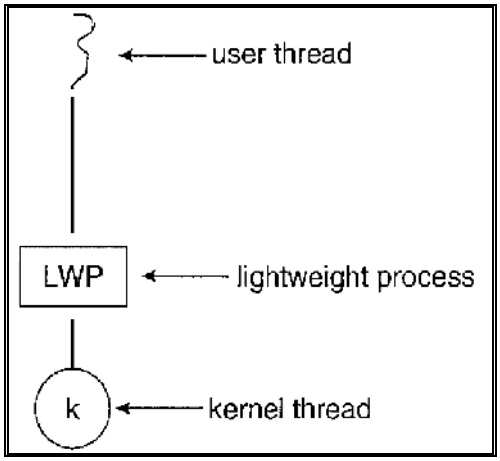


Fig 8.7 Lightweight Process (LWP)

To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks (such as while waiting for an i/o operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks. An application may require any number of LWPs to run efficiently.

8.6 THREAD SCHEDULING

User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).

8.6.1 CONTENTION SCOPE

Systems implementing many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP, a scheme known as ***process-contention scope (PCS)***, since competition for the CPU takes place among threads belonging to the same process.

To decide which kernel thread to schedule onto a CPU, the kernel uses system-contention scope (SCS). Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model, such as Windows XP, Solaris, and Linux, schedule threads using only SCS.

Typically, PCS is done according to priority-the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not

adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically prompt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.

8.6.2 PTHREAD SCHEDULING

Pthread API allows specifying either PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations. The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy. The Pthread IPC provides two functions for getting-and setting-the contention scope policy:

pthread_attr_setscope(pthread_attr_t *attr, int scope)
pthread_attr_getscope(pthread_attr_t *attr, int *scope)

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the pthread_attr_setscope() function is passed either the PTHREAD_SCOPE_SYSTEM or the PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set. In the case of pthread_attr_getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope. If an error occurs, each of these functions returns a non-zero value.

8.7 LET US SUM UP

- The creating process is called a parent process, whereas the new processes are called the children of that process.
- A process terminates when it finishes executing its final statement and asks the OS to delete it by using the exit system call.

- IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- If processes P and Q want to communicate, they must send messages to send and receive from each other; a communication link must exist between them.
- Message passing may be either blocking or non-blocking - also known as synchronous and asynchronous.
- There are mainly three types of multithreading models available for user and kernel threads.
- Generally six issues are considered for threading.

8.8 UNIT END QUESTIONS

1. Define :
   (a) Thread-specific Data
   (b) Multithread programming
   (c) Parent Process
2. Discuss different threading issues.
3. What is meant by cooperating processes?
4. Write a short note on Message Passing System.
5. How is a process created?

9

CLIENT-SERVER SYSTEMS

Unit Structure

9.0 OBJECTIVES

After reading this unit you will be able to:

- Describe the communication between Client-Server systems.
- Distinguish between various pipes.
- Learn Peterson's solution for achieving mutual exclusion
- Study the concept of Semaphores

9.1 INTRODUCTION

The message passing paradigm realizes exchange of information among processes without using shared memory. This feature makes it useful in diverse situations such as in communication between OS functionalities in a microkernel-based OS, in client–server computing, in higher-level protocols for communication, and in communication between tasks in a parallel or distributed program.

Amoeba is a distributed operating system developed at the Vrije Universiteit in the Netherlands during the 1980s. The primary goal of the Amoeba project is to build a transparent distributed operating system that would have the look and feel of a standard

time-sharing OS like UNIX. Another goal is to provide a test-bed for distributed and parallel programming.

Amoeba provides kernel-level threads and two communication protocols. One protocol supports the client–server communication model through remote procedure calls (RPCs), while the other protocol provides group communication. For actual message transmission, both these protocols use an underlying Internet protocol called the fast local Internet protocol (FLIP).

9.2 COMMUNICATION IN CLIENT-SERVER SYSTEM

Three strategies for communication in client-server systems are:

1. Sockets,
2. Remote procedure calls (RPCs), and
3. Pipes.

9.2.1 SOCKETS

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets-one for each process. A socket is identified by an IP address concatenated with a port number.

In general, sockets use a client-server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the com1ection. Servers implementing specific services (such as telnet, FTP, and I-HTP) listen to well-known ports (a telnet server listens to port 23; an FTP server listens to port 21; and a Web, or HTTP, server listens to port 80). All ports below 1024 are considered *·well known;* we can use them to implement standard services. When a client process initiates a request for a connection, it is assigned a port by its host computer. This port is some arbitrary number greater than 1024.

For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a Web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the Web server. This situation is illustrated in Figure 9.1. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

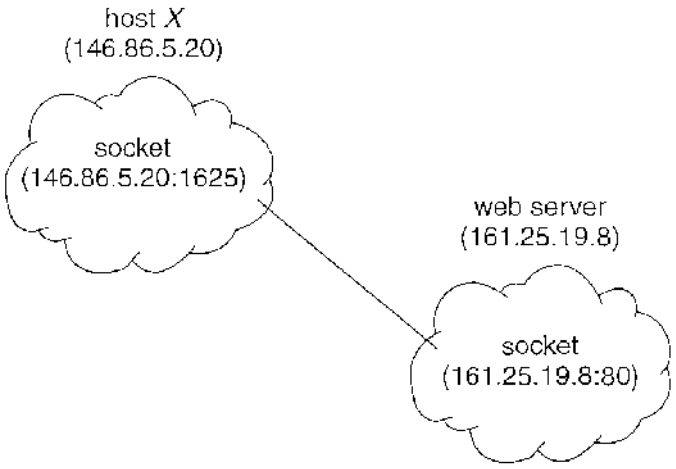


Fig 9.1 Communication using sockets

If another process also on host X wished to establish another connection with the same Web server, it would be assigned a port number greater than 1024 and not equal to 1625. This ensures all connections consist of a unique pair of sockets.

Let us illustrate sockets using Java, as it provides a much easier interface to sockets and has a rich library for networking utilities. Java provides three different types of sockets. Connection-oriented are implemented with the Socket class. Connectionless (UDP) use the DatagramSocket class. Finally, the MulticastSocket class is a subclass of the DatagramSocket class. A multicast socket allows data to be sent to multiple recipients.

Following example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could have any arbitrary number greater than 1024. When a cmmection is received, the server returns the date and time to the client. The date server is shown in Program Figure 9.2.

The server creates a ServerSocket that specifies it will listen to port 6013 and begins listening to the port with the accept() method. The server blocks on the accept() method waiting for a client to request a connection. When a connection request is received, accept() returns a socket that the server can use to communicate with the client.

The details of how the server communicates with the socket are as follows:

The server first establishes a PrintWriter object that it will use to communicate with the client allowing the server to write to the socket using the routine print() and println() methods for output. The

server process then sends the date to the client, calling the method println(). Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port on which the server is listening. We implement such a client in the Java program shown in Program Figure 9.3. The client creates a Socket and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal stream I/0 statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the loopback. When a computer refers to IP address 127.0.0.1 it is referring to itself.

This mechanism allows a client and server on the same host to communicate using the TCP /IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date server. In addition to an IP address an actual host name, such as *www.mu.ac.in* can be used as well.

```
import java.net.*;
import java.io.*;
public class DateServer{}
public static void main(String[] args) {
        try {}
}
ServerSocket sock= new ServerSocket(6013);
                        //now listen for connections
while (true) {}
Socket client= sock.accept();
PrintWriter pout = new
PrintWriter(client.getOutputStream(), true);
                        //write the Date to the socket
pout.println(new java.util.Date().toString());
                        //close the socket and resume
                        //listening for connections
client. close() ;
catch (IOException ioe) {
System.err.println(ioe);
}
```

Program Figure 9.2 Date server.

```
import java.net.*;
import java.io.*;
```

```
public class DateClient{}
public static void main(String[] args) {
        try {}
}
                        //make connection to server socket
Socket sock= new Socket("127.0.0.1",6013);
InputStream in= sock.getinputStream();
BufferedReader bin = new
BufferedReader(new InputStreamReader(in));
                        // read the date from the socket
String line;
while ( (line = bin.readLine()) !=null)
System.out.println(line);
                        // close the socket connection
sock. close() ;
catch (IDException ioe) {
System.err.println(ioe);
}
```

Program Figure 9.3 Date client.

9.2.2 Remote Procedure Calls

One of the most common forms of remote service is the RPC paradigm. The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections. The messages exchanged in RPC communication are well structured and no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message.

A *port* is simply a number included at the start of a message packet. Whereas a system normally has one network address, it can have many ports within that address to differentiate the many network services it supports. If a remote process needs a service, it addresses a message to the proper port. The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally. The RPC system hides the details that allow communication to take place by providing a on the client side.

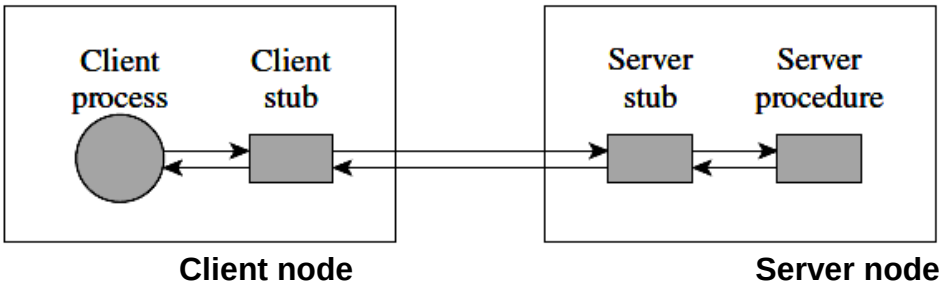


Fig 9.4 Overview of Remote Procedure Call (RPC)

Typically, a separate stub exists for each separate remote procedure. When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure. This stub locates the port on the server and *marshals* the parameters. Parameter marshalling involves packaging the parameters into a form that can be transmitted over a network. The stub then transmits a message to the server using message passing. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

One issue that must be dealt with concerns differences in data representation on the client and server machines. Consider the representation of 32-bit integers. Some systems (known as *big-endian)* store the most significant byte first, while other systems (known as *little-endian)* store the least significant byte first. Neither order is "better" per se; rather, the choice is arbitrary within a computer architecture. To resolve differences like this, many RPC systems define a machine-independent representation of data. One such representation is known as external data representation (XDR). On the client side, parameter marshalling involves converting the machine-dependent data into XDR before they are sent to the server. On the server side, the XDR data are unmarshalled and converted to the machine-dependent representation for the server.

The RPC scheme is useful in implementing a distributed file system. Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the distributed file system port on a server on which a file operation is to take place. The message contains the disk operation to be performed. The disk operation might be read, write, rename, delete, or status, corresponding to the usual file-related system calls. The return message contains any data resulting from that call, which is executed by the DFS daemon on behalf of the client.

9.2.3 Pipes

A pipe acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems and typically provide one of the simpler ways for processes to communicate with one another. Two common types of pipes used on both UNIX and Windows systems are *ordinary pipes* and *named pipes*.

Ordinary pipes allow two processes to communicate in standard producer-consumer fashion; the producer writes to one end of the (the write-end) and the consumer reads from the other end (the read-end).

In Named pipes, communication can be bidirectional, and no parent-child relationship is required. Once a named pipe is established, several processes can use it for communication. In fact, in a typical scenario, a named pipe has several writers. Additionally, named pipes continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes, although the details of implementation vary greatly. Named pipes are referred to as FIFOs in UNIX systems. On a Windows systems full-duplex communication is allowed, and the communicating processes may reside on either the same or different machines. Windows systems allow either byte- or message-oriented data.

9.3 THE CRITICAL SELECTION PROBLEM

Consider a system consisting n processes $\{P_0, P_1, \ldots P_{n-1}\}$. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

Thus, the execution of the critical section by the processes is mutually exclusive in time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

do {

entry section

critical section

exit section

remainder section
} while(1);

Program Fig 9.5 General structure of a typical process P_i

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

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

Mutual Exclusion:

If process Pi is executing in its critical section, then no other process can be executed in their critical sections.

Progress:

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

Bounded Waiting:

There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

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

The solutions do not rely on any assumption concerning the hardware instructions such as load, store, and test etc. or the number of processes that hardware supports.

9.4 PETERSON'S SOLUTION

In 1965, T. Dekker was the first one to device a software solution to mutual exclusion problem. Dekker's algorithm solves the

mutual exclusion problem but with a rather complex program. In 1981, Peterson discovered a much simpler algorithm for providing mutual exclusion. Thus rendering Dekker's solution obsolete.

Peterson's algorithm is shown below. This algorithm consists of two procedures written in ANSI C.

```
#include "prototype.h"

#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: who is entering (0 or
1) */
{
int other;                                  /* number of 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 (0 or 1)
*/
{
interested process = FALSE;                 /*indicate departure from
critical region */
```

Peterson's solution for achieving mutual exclusion

Before using the shared variables (i.e., before entering its critical region), each process calls enter_region with its own process number, 0 and 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the variables, the process calls leave_region to indicate that it is done and to allow the other process to enter, if it so desires.

Initially neither process is in its critical region. Now process 0 calls enter_region. It indicates its interest by setting its array element, and sets turn to 0. Since process1 is not interested, enter_region returns immediately. If process1 now calls enter_region, it will hang there until interested [0] goes to FALSE, an event that only happens when process 0 calls leave_region.

Now 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 lost. Suppose process 1 stores last, so turn is 1. When both processes come to the while statement, process 1 executes its zero times, and enters its critical region. Process 0 loops and does not enter its critical region.

9.5 SEMAPHORES

Definition:

Semaphore is a variable that has an integer value upon which the following three operations are defined:

1) A semaphore may be initialized to a non-negative value.
2) The wait operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked.
3) The signal operation increments the semaphore value. If the value is not positive, then a process blocked by a-wait operation is unblocked.

Concept of operation:

Two or more processes can cooperate by means of simple signals, such that the process can be forced to stop at a specified place until it has received a specific signal. For signaling, special variables called semaphores, a process executes the primitive wait(s); if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

The wait and signal primitives are assumed to be atomic; that is, they cannot be interrupted and each routine can be treated as an indivisible step.

```
Type semaphore =record
                    Count : integer;
                    Queue : list of process
                    End;
var s : semaphore;
wait (s) :
      s.count := s.count – 1;
```

```
        if s.count < 0
          then begin
               place this process in s . queue;
               block this process
               end;
signal (s) :
        s.count: = s.count + 1;
        if s.count ≤ 0
          then begin
              remove a process P from S.queue;
             place process P on ready list
        end;
```

Program Figure 9.6 Definition code for semaphores

Implementing mutual Exclusion semaphores:

```
program mutualexclusion;
const n = …; (*number of processes*);
var s:semaphore (:=1);
procedure P (I : integer);
begin
  repeat
  wait (s);
  <critical section>;
  signal (s);
  <remainder>
  forever
end
begin (*main program*)
  parbegin
    P(1);
    P(2);
    …
    P(n)
  parend
end.
```

Program Figure 9.7 Mutual Exclusion Using Semaphores

Figure shows a straightforward solution to the mutual exclusion problem by semaphores.

The semaphore is initialized to 1. Thus, the first process that executes a signal will be able to immediately enter the critical section, setting the value of s to 0. Another process attempting to enter the critical section will find it busy and will be blocked setting the value of s to – 1. Any number of processes may attempt to enter the section will find it busy and will be blocked. Any number of processes may attempt entry; each such unsuccessful attempt

results in a further decrement of the value of s. When the process that initially entered its critical section leaves, s is incremented and one of the blocked processes associated with the semaphore is put in a ready state. When it is next scheduled by the operating system, it may enter the critical section.

Instead of using an ordinary semaphores we can use a binary semaphores which is defined as in figure below:

A Binary semaphore may take on only value 0 and 1.

```
type semaphore = record
                    value: (0, 1);
                    queue:list of process
                   end;
var s : semaphore;
wait B(s) :
        if s.value = 1
            then
            s.value = 0
        else begin
           place this process in s.queue;
           block this process
        end;
signal B (s) :
    if s. count ≤ 0
        then
          if s. value : = 1
    else begin
       remove a process P from s.queue;
       place process P on ready list
    end;
```

Program Figure 9.8 Definition of Binary Semaphore Primitives

9.6 LET US SUM UP

- A socket is defined as an endpoint for communication
- The RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections.
- A port is simply a number included at the start of a message packet.

- When the client invokes a remote procedure, the RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure
- A pipe acts as a conduit allowing two processes to communicate.
- Two common types of pipes used on both UNIX and Windows systems are ordinary pipes and named pipes
- The critical-section problem is to design a protocol that the processes can use to cooperate
- In 1965, T. Dekker was the first one to device a software solution to mutual exclusion problem
- In 1981, Peterson discovered a much simpler algorithm for providing mutual exclusion
- Semaphore is a variable that has an integer value upon which the following three operations are defined:

  1. A semaphore may be initialized to a non-negative value.
  2. The wait operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked.
  3. The signal operation increments the semaphore value. If the value is not positive, then a process blocked by a-wait operation is unblocked.

9.7 UNIT END QUESTIONS

1. Define : (a) Socket (b) Pipe (c) Semaphore
2. What is RPC?
3. Discuss in detail the Critical-Section Problem.
4. Write a brief note on Peterson's solution.
5. Describe the architecture of semaphore with the help of program figures.

10

MAIN MEMORY

Unit Structure

10.0 OBJECTIVES

After reading this unit you will be able to:

- Manage memory with/without swapping and paging
- Study various memory allocation techniques
- Define paging
- Define Segmentation

10.1 INTRODUCTION

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

A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed Depending on the memory management in use

the process may be moved between disk and memory during its execution. The collection of processes on the disk that are waiting to be brought into memory for execution forms the input queue. i.e. selected one of the process in the input queue and to load that process into memory.

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

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

• Load time: If it is not known at compile time where the process will reside in memory, then the compiler must generate re-locatable code.

• Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

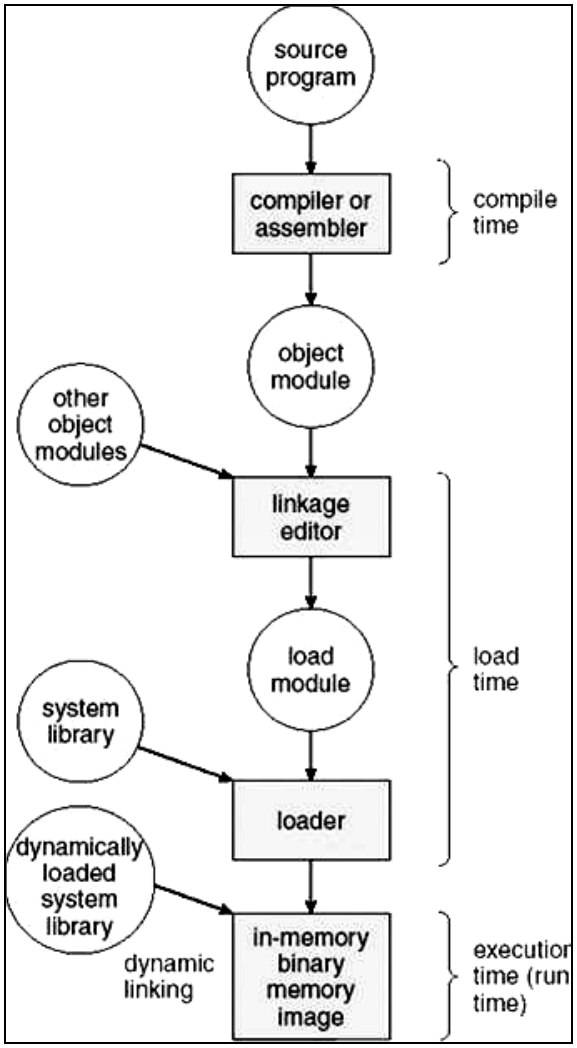


Fig 10.1 BINDING OF INSTRUCTIONS

10.2 MEMORY MANAGEMENT WITHOUT SWAPPING OR PAGING

10.2.1 DYNAMIC LOADING

Better memory-space utilization can be done by dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed. The advantage of dynamic loading is that an unused routine is never loaded.

10.2.2 DYNAMIC LINKING

Most operating systems support only static linking, in which system language libraries are treated like any other object module and are combined by the leader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. With dynamic linking, a stub is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routing.

10.2.3 OVERLAYS

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

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

Let us consider

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

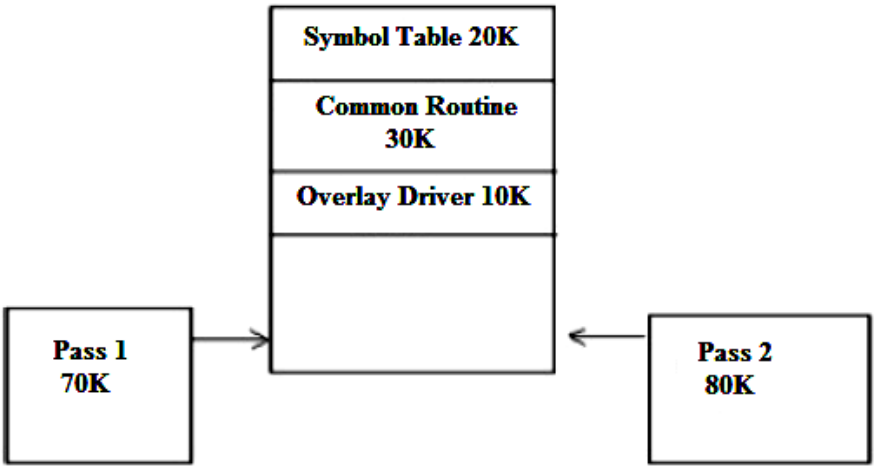


FIG 10.2 OVERLAYS

To load everything at once, we would require 200K of memory. If only 150K is available, we cannot run our process. But pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2.

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

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

10.2.4 LOGICAL VERSUS PHYSICAL ADDRESS SPACE

An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit is commonly referred to as a physical address.

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

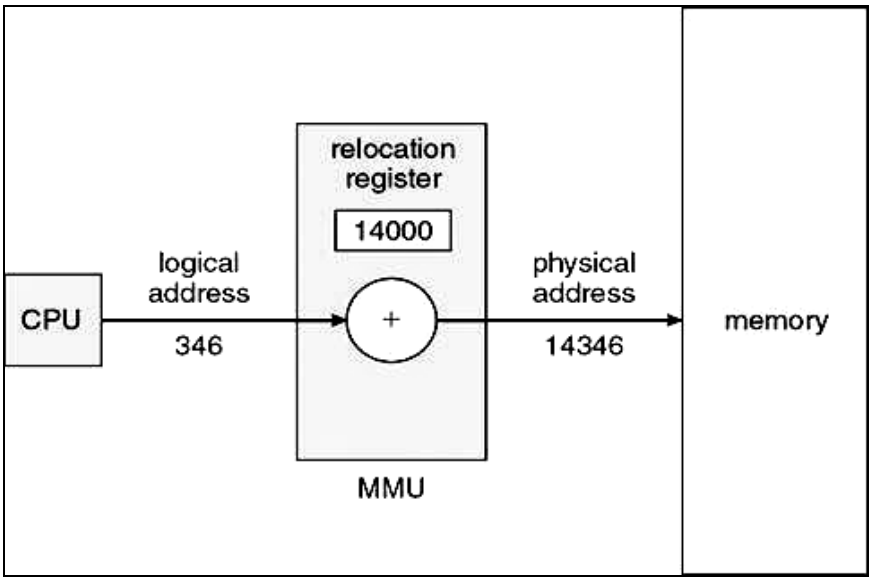


FIG 10.3 DYNAMIC RELOCATION USING RELOCATION REGISTER

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

The base register is called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 13000, then an attempt by the user to address location 0 dynamically relocated to location 13,000; an access to location 347 is mapped to location 13347. The MS-DOS operating system running on the Intel 80x86 family of processors uses four relocation registers when loading and running processes.

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

The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses.

Logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R). The user generates only logical addresses.

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

10.3 SWAPPING

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. Assume a multiprogramming environment with a round robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed. When each process finishes its quantum, it will be swapped with another process.

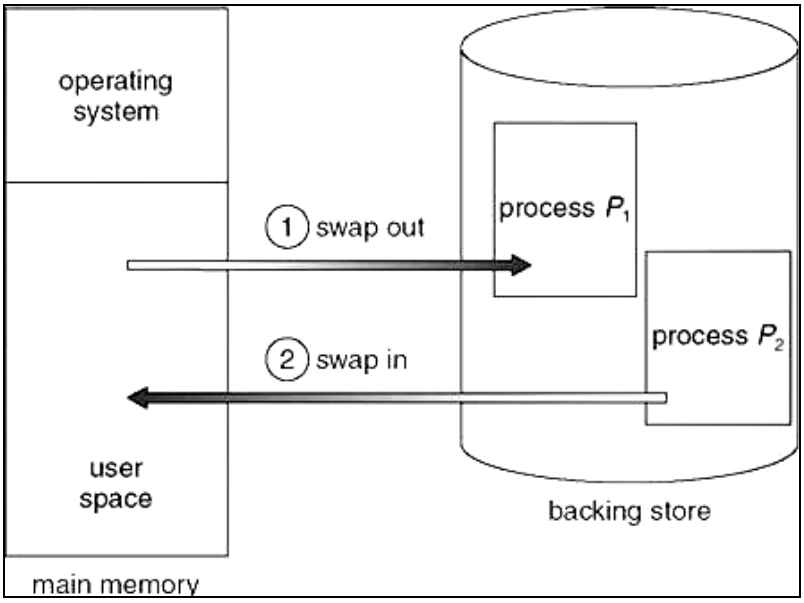


FIG 10.4 SWAPPING OF TWO PROCESSES USING A DISK AND A BACKING STORE

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called rollout, roll in.

A process swapped out will be swapped back into the same memory space that it occupies previously. If binding is done at assembly or load time, then the process cannot be moved to different location. If execution-time binding is being used, then it is possible to swap a process into a different memory space.

Swapping requires a backing store. The backing store is commonly a fast disk. It is large enough to accommodate copies of all memory images for all users. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.

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

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

10.4 CONTIGUOUS MEMORY ALLOCATION

The main memory must accommodate both the operating system and the various user processes. The memory is usually divided into two partitions, one for the resident operating system, and one for the user processes. To place the operating system in low memory, we shall discuss only me situation where the operating system resides in low memory. The development of the other situation is similar. Common Operating System is placed in low memory.

10.4.1 SINGLE-PARTITION ALLOCATION

If the operating system is residing in low memory, and the user processes are executing in high memory. And operating-system code and data are protected from changes by the user processes. We also need protect the user processes from one another. We can provide this 2 protection by using a relocation registers.

The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100,040 and limit =74,600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory.

The relocation-register scheme provides an effective way to allow the operating-system size to change dynamically.

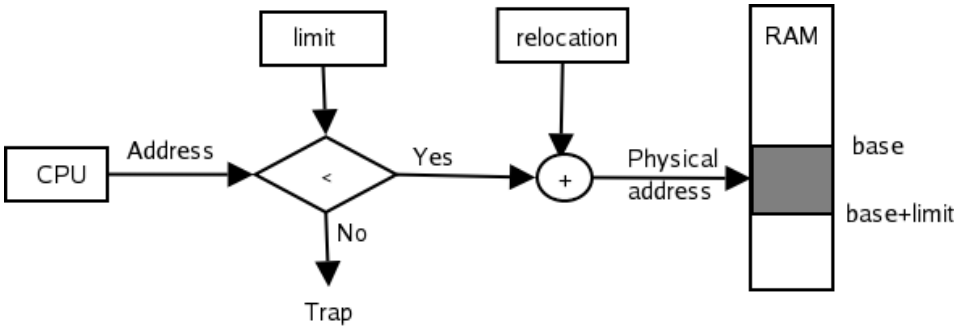
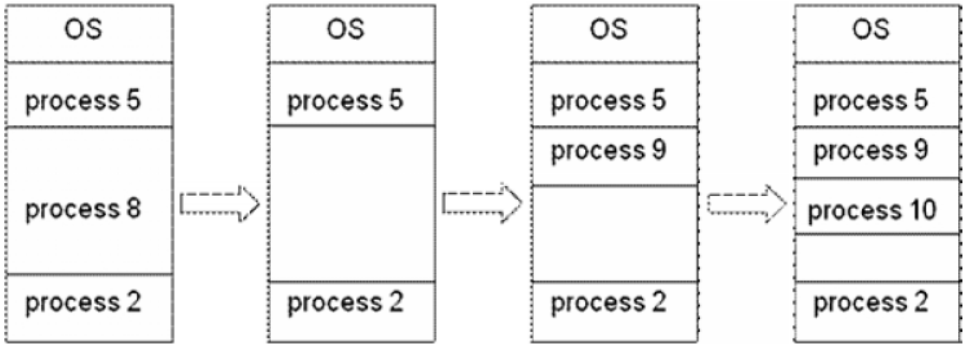


Fig 10.5 HARDWARE SUPPORT FOR RELOCATION AND LIMIT REGISTERS

10.4.2 MULTIPLE-PARTITION ALLOCATION

One of the simplest schemes for memory allocation is to divide memory into a number of fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound of partitions. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block, of available memory, a hole. When a process arrives and needs memory, operating system forms a hole large enough for this process.



When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, we merge these adjacent holes to form one larger hole.

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

First-fit:

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

Best-fit:

Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy-produces the smallest leftover hole.

Worst-fit:

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

10.4.3 EXTERNAL AND INTERNAL FRAGMENTATION

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when enough to the memory space exists to satisfy a request, but it is not contiguous; storage is fragmented into a large number of small holes. Depending on the total amount of memory storage and the average process size, external fragmentation may be either a minor or a major problem. Given N allocated blocks, another 0.5N blocks will be lost due to fragmentation. That is, one-third of memory may be unusable. This property is known as the 50-percent rule.

Internal fragmentation – memory that is internal to partition, but is not being used.

10.5 PAGING

External fragmentation is avoided by using paging. Physical memory is broken into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames. Every address generated by the CPU is divided into any two parts: a page number (p) and a page offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the gage offset to define the physical memory address that is sent to the memory unit.

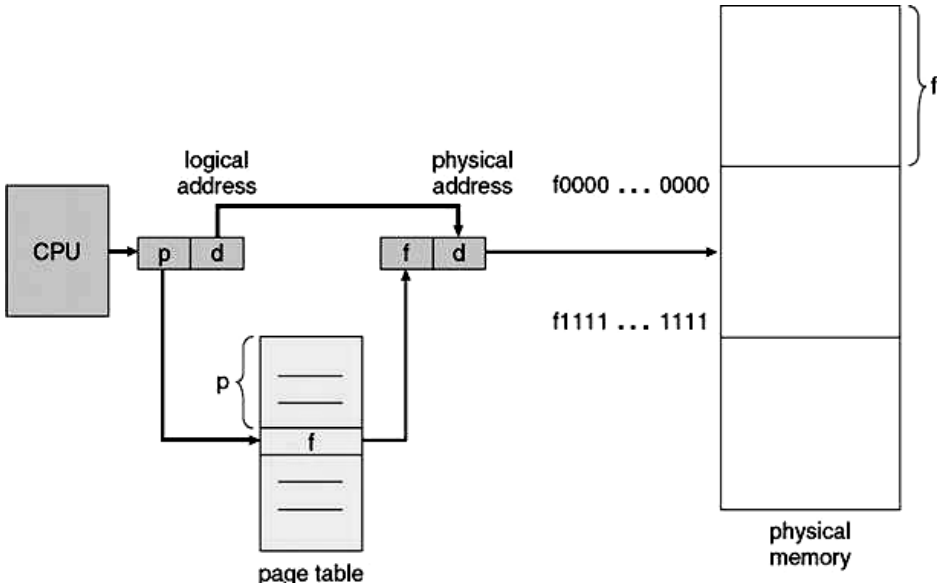


Fig 10.6 Paging Hardware

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

page number	page offset
p	d
m-n	n

where p is an index into the page table and d is the displacement within page.

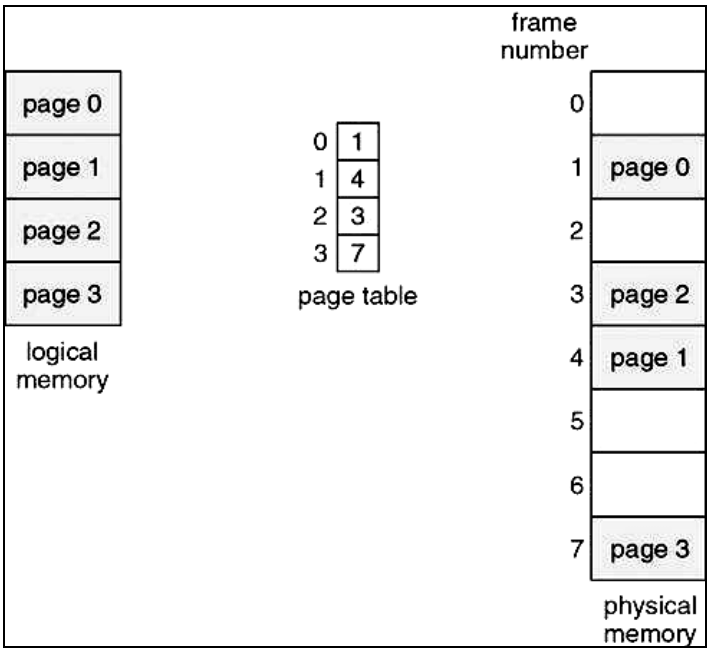


Fig 10.7 Paging model of logical and physical memory

Paging is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Any free frame can be allocated to a process that needs it. If process size is independent of page size, we can have internal fragmentation to average one-half page per process.

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

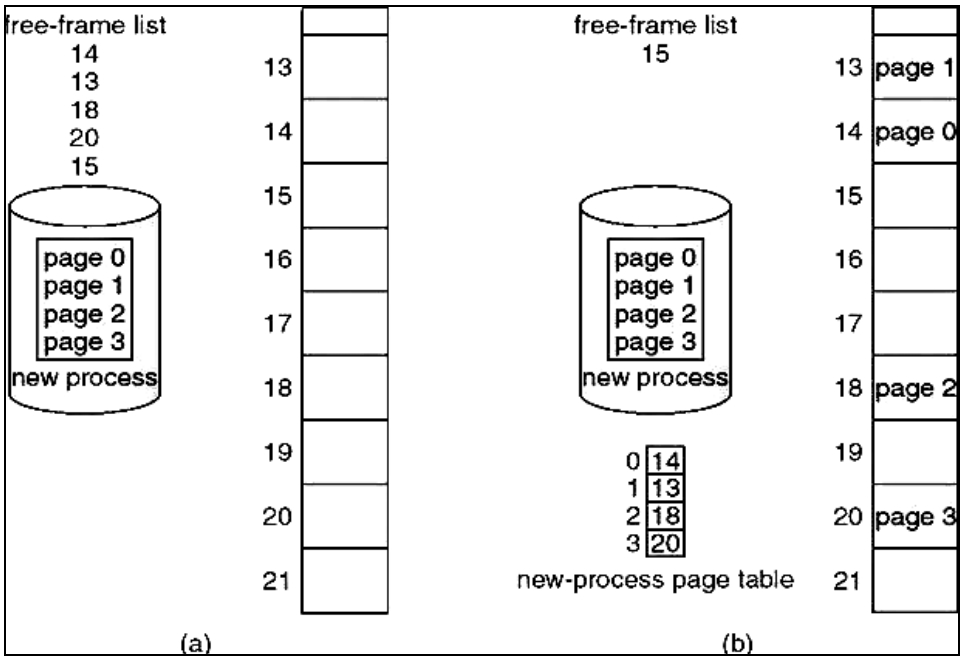


FIG 10.8 FREE FRAMES
(a) BEFORE ALLOCATION (b) AFTER ALLOCATION

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

The operating system is managing physical memory, it must be aware of the allocation details of physical memory: which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free allocated and, if it is allocated, to which page of which process or processes. The operating system maintains a copy of the page table for each process. Paging therefore increases the context-switch time.

10.6 SEGMENTATION

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of **segments**. It is not required that all segments of all programs be of the same length, although there is a maximum segment length. As with paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

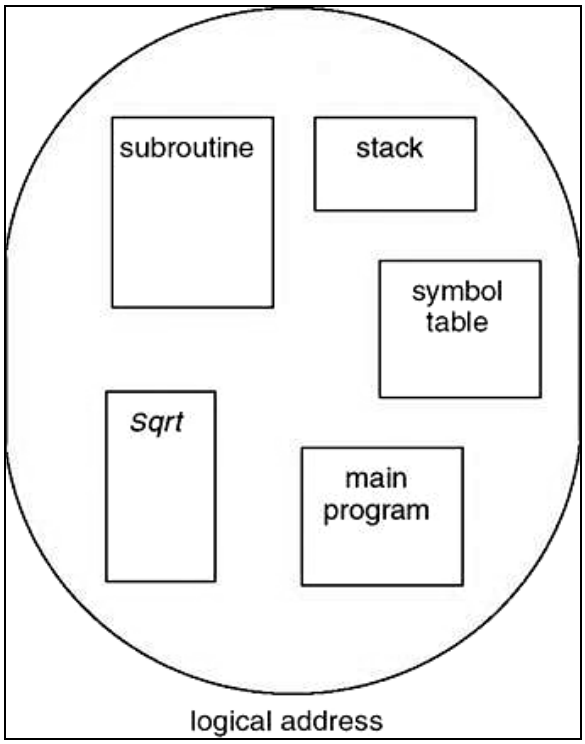


Fig 10.9 USER'S VIEW OF A PROGRAM

Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning. In segmentation, a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken into a number of smaller pieces, the external fragmentation should less. Whereas paging is invisible to the programmer, segmentation usually visible and is provided as a convenience for organizing programs and data.

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

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

- Extract the segment number as the leftmost n bits of the logical address.

- Use the segment number as an index into the process segment table to find the starting physical address of the segment.
- Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
- The desired physical address is the sum of the starting physical address of the segment plus the offset.

Segmentation and paging can be combined to have a good result.

10.7 LET US SUM UP

- This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routing
- The execution-time address-binding scheme results in an environment where the logical and physical addresses differ
- The base register is called a relocation register.
- Every address generated by the CPU is divided into any two parts: a page number (p) and a page offset (d)
- The operating system maintains a copy of the page table for each process
- A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of segments

10.8 UNIT END QUESTIONS

1. What is dynamic linking?
2. Describe overlays in brief.
3. What is the difference between single-partition allocation and multiple-partition allocation?
4. Write short notes on :
    a. Paging
    b. Segmentation

11

VIRTUAL MEMORY

Unit Structure

11.0 OBJECTIVES

After reading this unit you will be able to:

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging & page-replacement algorithms, and allocation of page frames.
- To discuss the principle of the working-set model

11.1 INTRODUCTION

Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory. This separation allows an

extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

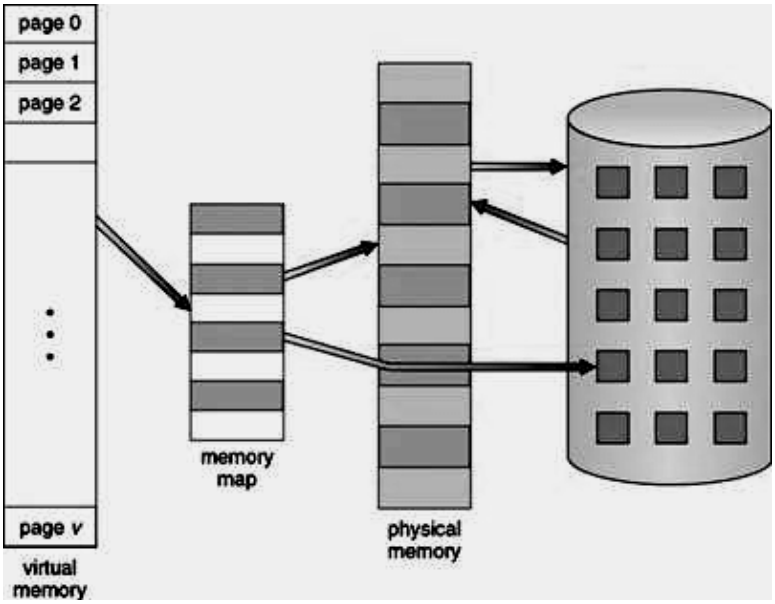


Fig 11.1 Virtual memory is larger than physical memory

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

11.2 DEMAND PAGING

A demand paging is similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory. When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

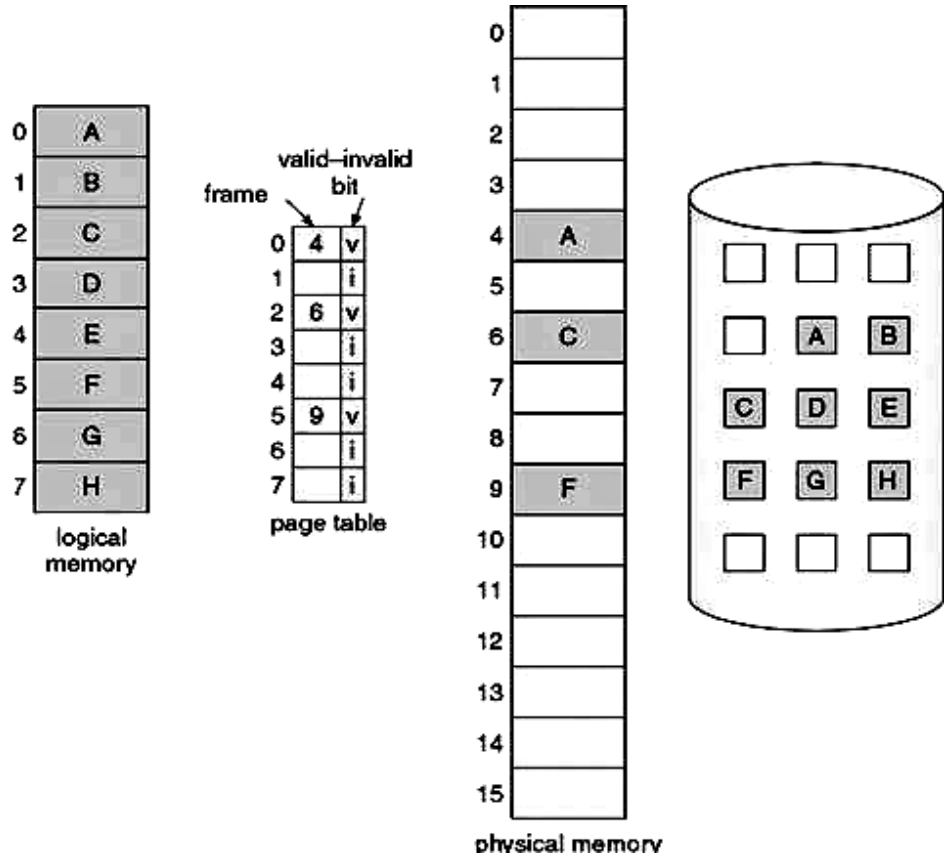


Fig 11.2 Transfer of a paged memory to contiguous disk space

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme. Where valid and invalid pages can be checked checking the bit and marking a page will have no effect if the process never attempts to access the pages. While the process executes and accesses pages that are memory resident, execution proceeds normally. Access to a page marked invalid causes a page-fault trap. This trap is the result of the operating system's failure to bring the desired page into memory. But page fault can be handled as following:

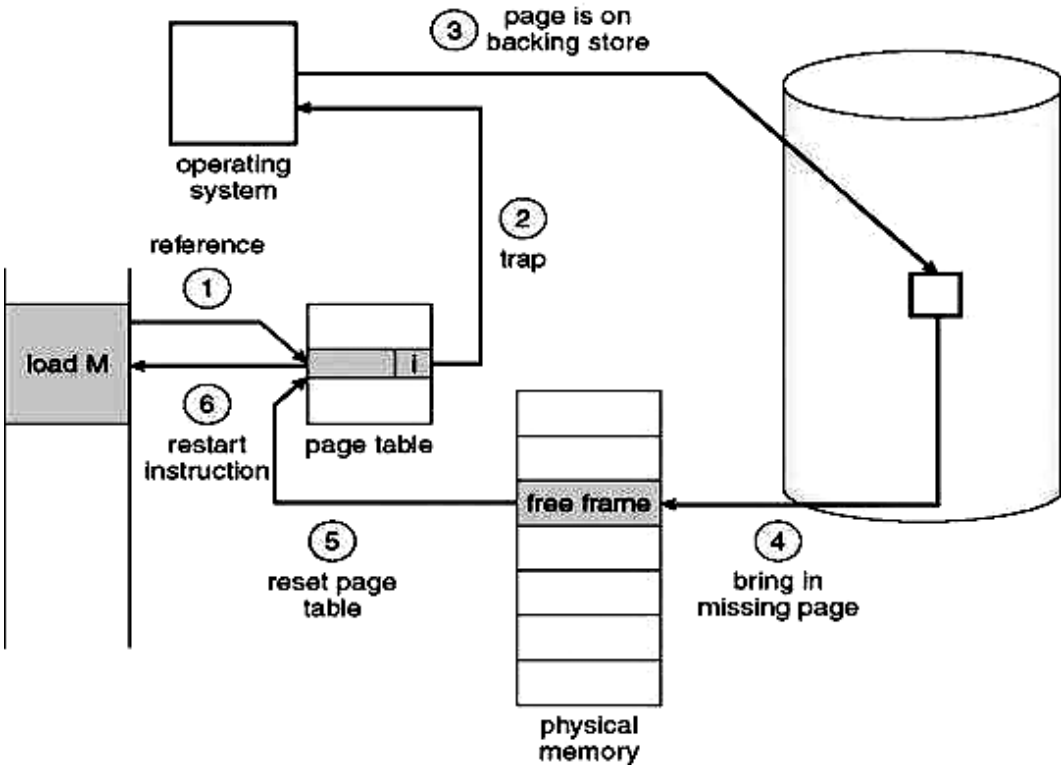


Fig 11.3 Steps in handling a page fault

1. Check an internal table for this process to determine whether the reference was a valid or invalid memory access.
2, If the reference was invalid, terminate the process. If it was valid, but not yet brought in that page, we now page in the latter
3. Find a free frame
4. Schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been memory.

Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory. The page replacement is used to make the frame free if they are not in used. If no frame is free then other process is called in.

11.3 PAGE REPLACEMENT ALGORITHMS

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

1. For a given page size we need to consider only the page number, not the entire address.

2. If we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

Eg:- Consider the address sequence 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0104, 0101, 0609, 0102, 0105 and reduce to 1, 4, 1,6,1, 6, 1, 6, 1,6, 1

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

11.3.1 FIFO Algorithm

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory.

The first three references (7, 0, 1) cause page faults, and are brought into these empty eg. 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1 and consider 3 frames. This replacement means that the next reference to 0 will fault. Page 1 is then replaced by page 0.

Reference string

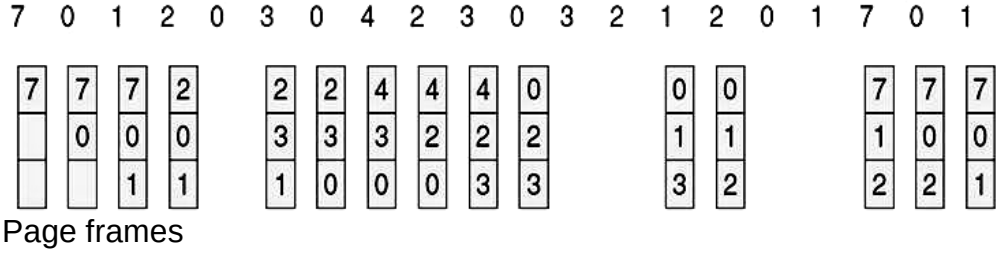
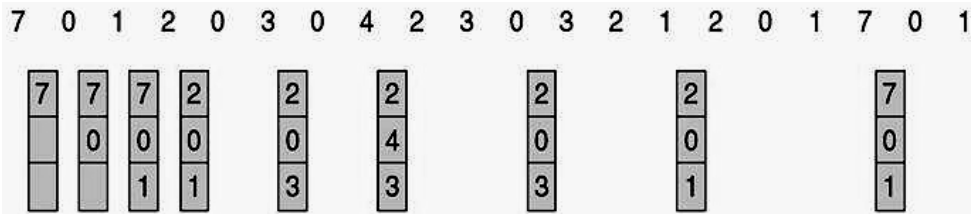


Fig 11.4 FIFO Page-replacement algorithm

11.3.2 Optimal Algorithm

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

Reference String



Page frames

Fig 11.5 Optimal Algorithm

Now consider the same string with 3 empty frames.

The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. Optimal replacement is much better than a FIFO.

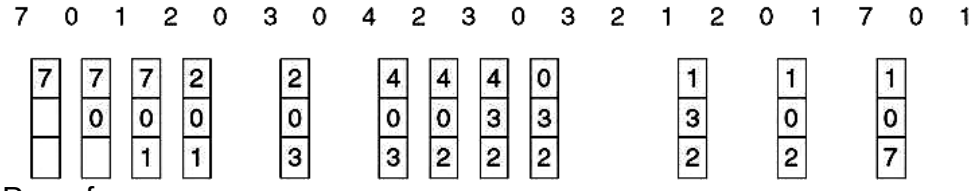
The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

11.3.3 LRU Algorithm

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

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

Reference String



Page frames

Fig 11.6 LRU Algorithm

Let S^R be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on S^R.

11.3.4 LRU Approximation Algorithms

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

(i) Additional-Reference-Bits Algorithm

The operating system shifts the reference bit for each page into the high-order or of its 8-bit byte, shifting the other bits right 1 bit, discarding the lower order bit. These 8-bit shift registers contain the history of page use for the last eight, time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value 11111111.

(ii) Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page gets a second chance, its reference bit is cleared and its arrival e is reset to the current time.

(iii)Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering troth the reference bit and the modify bit as an ordered pair.

1. (0,0) neither recently used nor modified — best page to replace
2. (0,1) not recently used but modified — not quite as good, because the page will need to be written out before replacement
3. (1,0) recently used but clean — probably will be used again soon
4. (1,1) recently used and codified - probably will be'iTsed aaain, and write out will be needed before replacing it

(iv) Counting Algorithms

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

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

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

11.3.5 Page Buffering Algorithm

When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool. When the FIFO replacement algorithm mistakenly replaces a page mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm.

11.4 MODELING PAGING ALGORITHM

11.4.1 WORKING-SET MODEL

The working-set model is based on the assumption of locality.

- Δ defines the working-set window: some # of memory references
- Examine the most recent Δ page references.
- The set of pages in the most recent Δ is the *working set* or an approximation of the program's locality.

PAGE REFERENCE TABLE

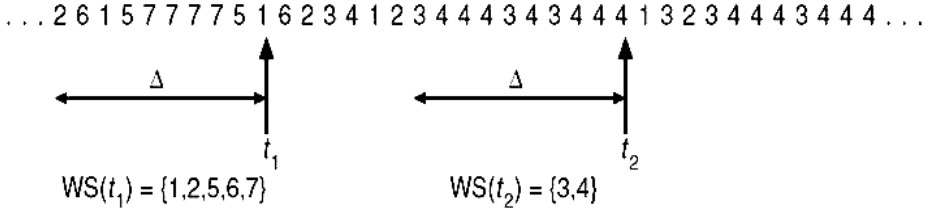


Fig 11.7 Working-set model

- The accuracy of the working set depends on the selection of Δ.
- If Δ is too small, it will not encompass the entire locality
- If Δ is too large, it may overlap several localities.
- If Δ is &infinity; the working set is the set of all pages touched during process execution
- WSS_i is working set size for process p_i
- $D = \Sigma\ WSS_i$, where D is the total *Demand* from frames
- If $D > m$, then thrashing will occur, because some processes will not have enough frames

  Using the working-set strategy is simple:

- The OS monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size.
- If there are enough extra frames, a new process can be initiated.
- If the sum of the working set sizes increases, exceeding the total number of available frames, the OS selects a process to suspend.
- The working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible and optimizes CPU utilization.

11.5 DESIGN ISSUES FOR PAGING SYSTEMS

11.5.1 Prepaging

Prepaging is an attempt to prevent high level of initial paging. In other words, prepaging is done to reduce the large number of page faults that occurs at process startup.
Note: Prepage all or some of the pages a process will need, before they are referenced.

11.5.2 Page Size

Page size selection must take into consideration:

· fragmentation
· table size
· I/O overhead
· locality

11.5.3 Program structure

int [128,128] data;
Each row is stored in one page

Program 1:

```
for (j = 0; j <128; j++)
  for (i = 0; i < 128; i++)
      data[i,j] = 0;
```

128 x 128 = 16,384 page faults

Program 2:

```
for (i = 0; i < 128; i++)
  for (j = 0; j < 128; j++)
      data[i,j] = 0;
```

128 page faults

11.5.4 I/O Interlock

Pages must sometimes be locked into memory. Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

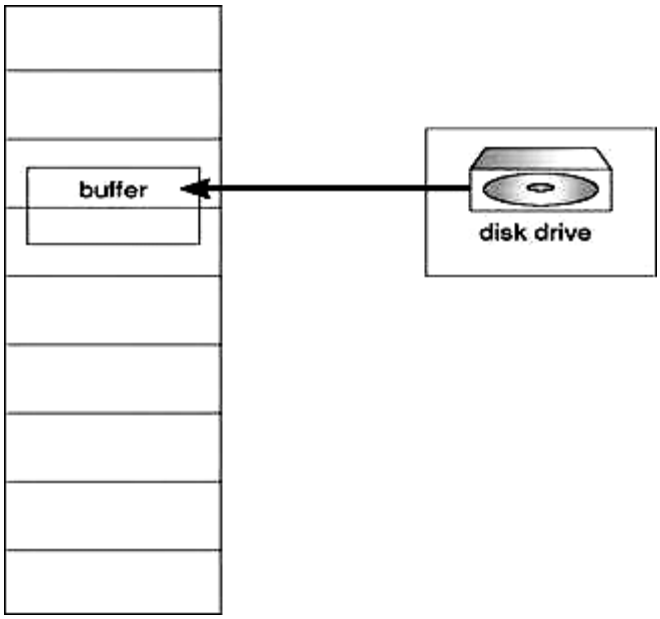


Fig 11.8 The reason why frames used for 1/0 must be in memory

11.6 LET US SUM UP

- Virtual memory is the separation of user logical memory from physical memory.
- Virtual memory is commonly implemented by demand paging.
- A demand paging is similar to a paging system with swapping.
- The page replacement is used to make the frame free if they are not in used.
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- LRU replacement associates with each page the time of that page's last use.
- When the FIFO replacement algorithm mistakenly replaces a page mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary.

- The working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible and optimizes CPU utilization.

11.7 UNIT END QUESTIONS

1. What is Virtual Memory?
2. Describe FIFO Algorithm in detail.
3. Write a short note on working-set model.
4. Briefly discuss various design issues for paging system.

12

FILE SYSTEM INTERFACE AND IMPLEMENTATION

Unit Structure

12.0 OBJECTIVES

After reading this unit you will be able to:

- Describe various allocation methods
- Manage free space
- Study various File sharing systems

12.1 INTRODUCTION

The most important parts of an operating system is the file system. The file system provides the resource abstractions typically associated with secondary storage. The file system permits users to create data collections, called files, with desirable properties, such as the following:

- **Long-term existence**: Files are stored on disk or other secondary storage and do not disappear when a user logs off.
- **Sharable between processes**: Files have names and can have associated access permissions that permit controlled sharing.
- **Structure:** Depending on the file system, a file can have an internal structure that is convenient for particular applications. In addition, files can be organized into hierarchical or more complex structure to reflect the relationships among files.

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

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

Typically, a file system maintains a set of attributes associated with the file.

12.2 FILE CONCEPTS

12.2.1 FILE STRUCTURE

Three terms are used for files

- Field
- Record

- Database

A field is the basic element of data. An individual field contains a single value.

A record is a collection of related fields that can be treated as a unit by some application program.

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

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

12.2.2 FILE MANAGEMENT SYSTEMS

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

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

File System Architecture

At the lowest level, device drivers communicate directly with peripheral devices or their controllers or channels. A device driver is responsible for starting I/O operations on a device and processing the completion of an I/O request. For file operations, the typical devices controlled are disk and tape drives. Device drivers are usually considered to be part of the operating system.

The next level is referred to as the basic file system or the physical I/O level. This is the primary interface with the environment outside of the computer system. It deals with blocks of data that are exchanged with disk or tape system.

The basic I/O supervisor is responsible for all file I/O initiation and termination. At this level control structures are maintained that deal with device I/O scheduling and file status. The basic 1.0 supervisor selects the device on which file I/O is to be performed, based on the particular file selected.

Logical I/O enables users and applications to access records. The level on the file system closest to the user is often termed the access method. It provides a standard interface between applications and the file systems device that hold the data. Different access methods reflect different file structures and different ways of accessing and processing the data.

The I/O control, consists of device drivers and interrupt handlers to transfer information between the memory and the disk system. A device driver can be thought of as a translator.

The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

File organization and access:

The term file organization to refer to the logical structuring of the records as determined by the way in which they are accessed. The physical organization of the file on secondary storage depends on the blocking strategy and the file allocation strategy. In choosing a file organization.

1. Short access time
2. Ease of update
3. Economy of storage
4. Simple maintenance
5. Reliability.

The relative priority of these criteria will depend on the applications that will use the file.

The file-organization module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 (or 1) through N, whereas the physical blocks containing the data usually do not match the logical numbers, so a translation is needed to locate each block. The file-organization module also includes the free-space manager, which

tracks unallocated and provides these blocks to the file organization module when requested.

The logical file system uses the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. The logical file system is also responsible for protection and security.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create new file, it reads the appropriate directory into memory, updates it with the new entry, and writes it back to the disk.

Once, the file is found, the associated information such as size, owner, and data block locations are generally copied into a table in memory, referred to as the open-file table; consisting of information about all the currently opened files.

The first reference to a file (normally an open) causes the directory structure to be searched and the directory entry for this file to be copied into the table of opened files. The index into this table is returned to the user program, and all further references are made through the index rather than with a symbolic name. The name given to the index varies. UNIX systems refer to it as a file descriptor, Windows/NT as a file handle, and other systems as a file control block. Consequently, as long as the file is not closed, all file operations are done on the open-file table. When the file is closed by all users that have opened it, the updated file information is copied back to the disk-based directory structure.

12.3 FILE SYSTEM MOUNTING

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

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

12.3.1 Allocation Methods

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

12.3.1.1 Contiguous Allocation

The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. Notice that with this ordering assuming that only one job is accessing the disk, accessing block b+1 after block b normally requires no head movement. When head movement is needed, it is only one track. Thus, the number of disk seeks required for accessing contiguous allocated files is minimal.

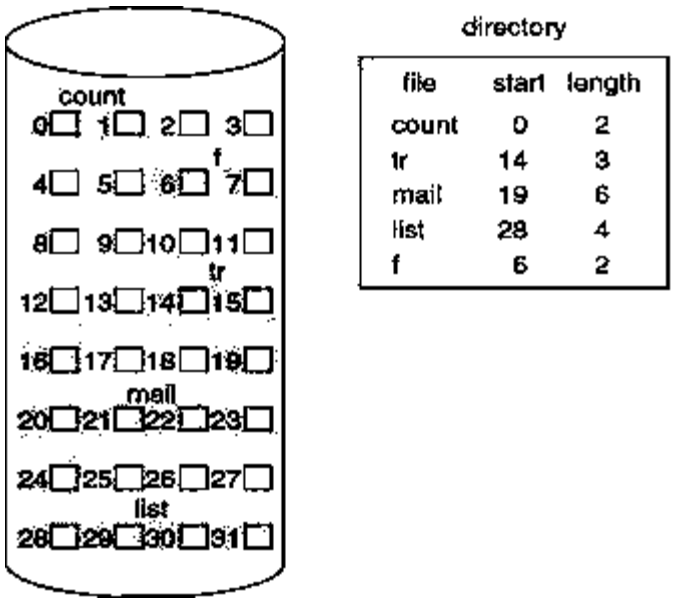


Fig 12.1 Contiguous allocation of disk space

Contiguous allocation of a tile is defined by the disk address and length (in block units) of the first block. If the file is n blocks long, and starts at a specific location then it occupies blocks b, b + 1, b +2,..,b+n-1. The directory entry for each file indicates the address of the starting block and the length of the area allocate for this file.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block b+i.

The contiguous disk-space-allocation problem can be seen to be a particular application of the general dynamic storage-allocation First Fit and Best Fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown

that both first-fit and best-fit are more efficient than worst-fit in terms of both time and storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.

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

To prevent loss of significant amounts of disk space to external fragmentation, the user had to run repacking routine that copied the entire file system onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk by allocating contiguous space from this one large hole. This scheme effectively compacts all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time. The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. During this down time, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines.

A major problem is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. The user will normally over estimate the amount of space needed, resulting in considerable wasted space.

12.3.1.2 Linked Allocation

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

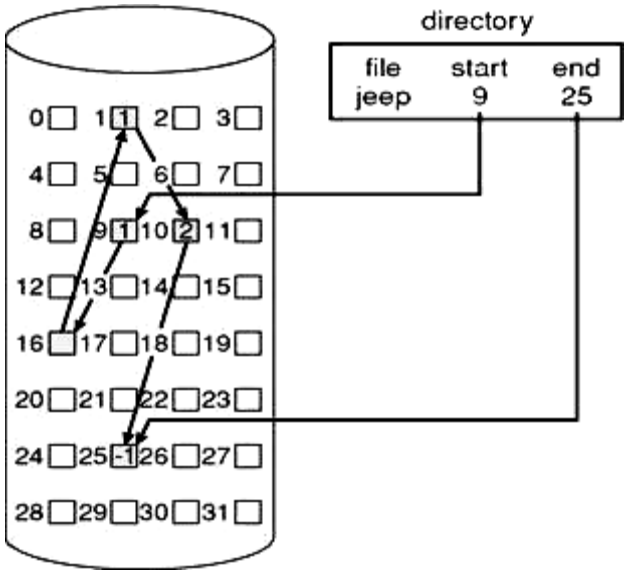


Fig 12.2 Linked allocation of disk space

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

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

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

The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate the clusters rather than blocks. For instance, the file system define a cluster as 4 blocks and operate on the disk in only cluster units. Pointers then use a much smaller percentage of the file's disk space. This method allows the logical-to-physical block mapping to remain simple, but improves disk throughput (fewer disk head seeks) and decreases the space needed for block allocation and free-list management. The cost of this approach an increase in internal fragmentation.

Yet another problem is reliability. Since the files are linked together by pointers scattered all over the disk consider what would happen it a pointer— were lost or damaged. Partial solutions are to use doubly linked iists or to store the file name and relative block

number in each block; however, these schemes require even more overhead for each file.

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

12.3.1.3 Indexed Allocation

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

Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block.

When the file is created, all pointers in the index block are set to nil. When the ith block is first written, a block is obtained: from the free space manager, and its address- is put in the ith index-block entry.

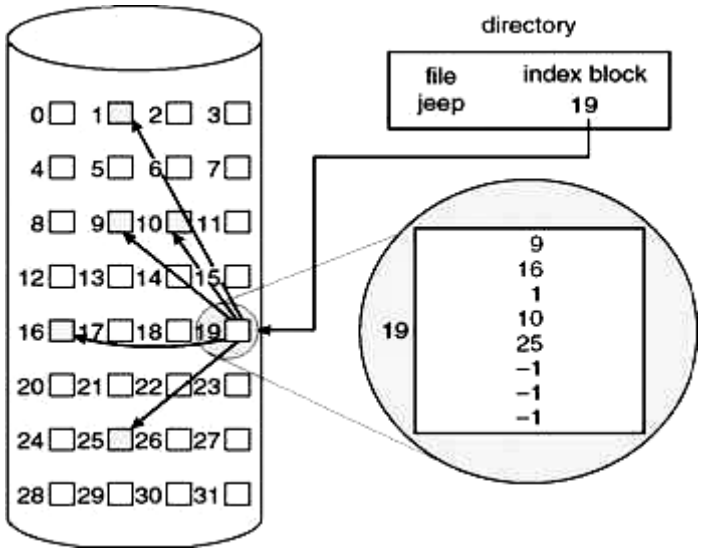


Fig 12.3 Indexed allocation of disk space

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

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

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

12.4 FREE SPACE MANAGEMENT

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

12.4.1 BIT VECTOR

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

00111100111111000110000011100000 .....

The main advantage of this approach is that it is relatively simple and efficient to find the first free block or n consecutive free

blocks on the disk. The calculation of the block number is (number of bits per word) x (number of 0-value words) + offset of first 1 bit

12.4.2 LINKED LIST

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

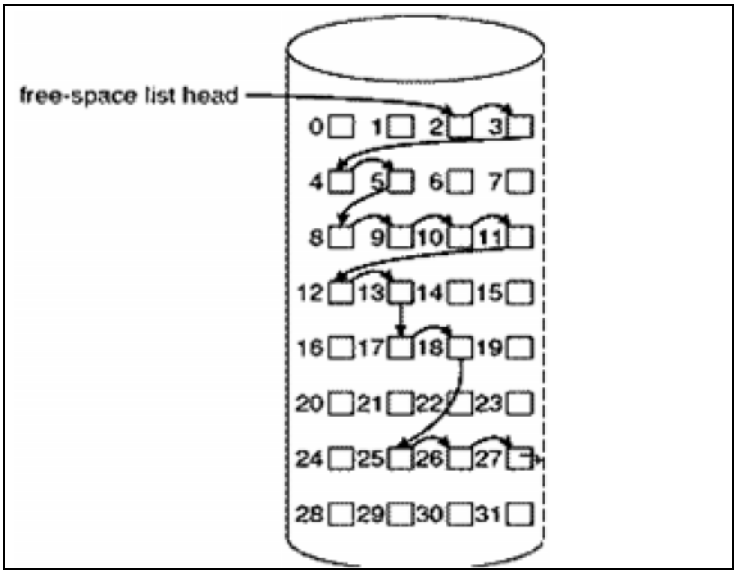


Fig 12.4 Linked free-space list on disk

12.4.3 GROUPING

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

12.4.4 COUNTING

Several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous allocation algorithm or through clustering. A list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block A. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a

simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

12.5 FILE SHARING

Once multiple users are allowed to share files, the challenge is to extend sharing to multiple file systems, including remote file systems.

12.5.1 MULTIPLE USERS

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system. Although many approaches have been taken to meet this requirement, most system have evolved to use the concepts of file (or directory) owner (or user) and group. The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file. For example, the owner of a file on a UNIX system can issue all operations on a file, while members of the file's group can execute one subset of those operations, and all other users can execute another subset of operations. Exactly which operations can be executed by group members and other users is definable by the file's owner.

12.5.2 REMOTE FILE SYSTEMS

Remote File system uses networking to allow file system access between systems:

- Manually via programs like **FTP**
- Automatically, seamlessly using **distributed file systems**
- Semi automatically via the **world wide web**

12.5.3 CONSISTENCY SEMANTICS

Consistency semantics represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. In particular, they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

12.6 NFS

Network File Systems [NFS] are standard UNIX client-server file sharing protocol. Interconnected workstations are viewed as a

set of independent machines with independent file systems, which allows sharing among these file systems in a transparent manner

A remote directory is mounted over a local file system directory. The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory. Files in the remote directory can then be accessed in a transparent manner.

Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory.

NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media. This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces.

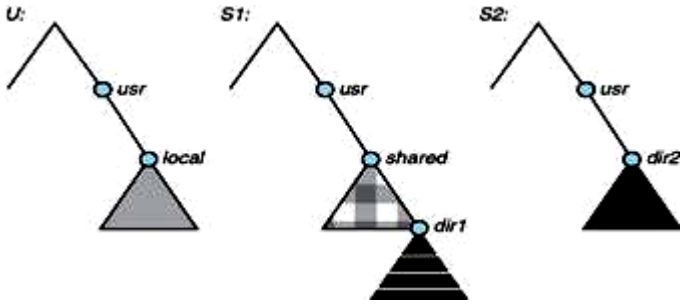


Fig 12.5 Three Independent File Systems

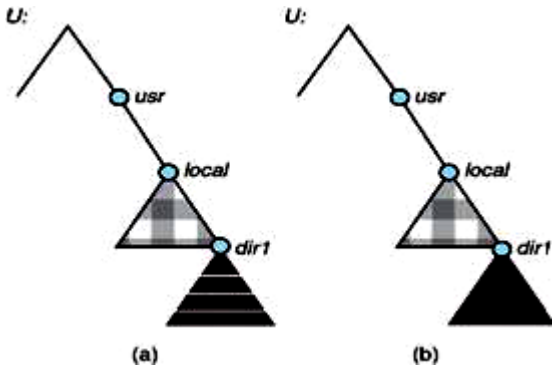


Fig 12.6 Mounting in NFS
(a) Mounts (b) Cascading Mounts

NFS Protocol

NFS protocol provides a set of remote procedure calls for remote file operations. The procedures support the following operations:

- searching for a file within a directory

- reading a set of directory entries
- manipulating links and directories
- accessing file attributes
- reading and writing files

NFS servers are stateless; each request has to provide a full set of arguments. Modified data must be committed to the server's disk before results are returned to the client. The NFS protocol does not provide concurrency-control mechanisms

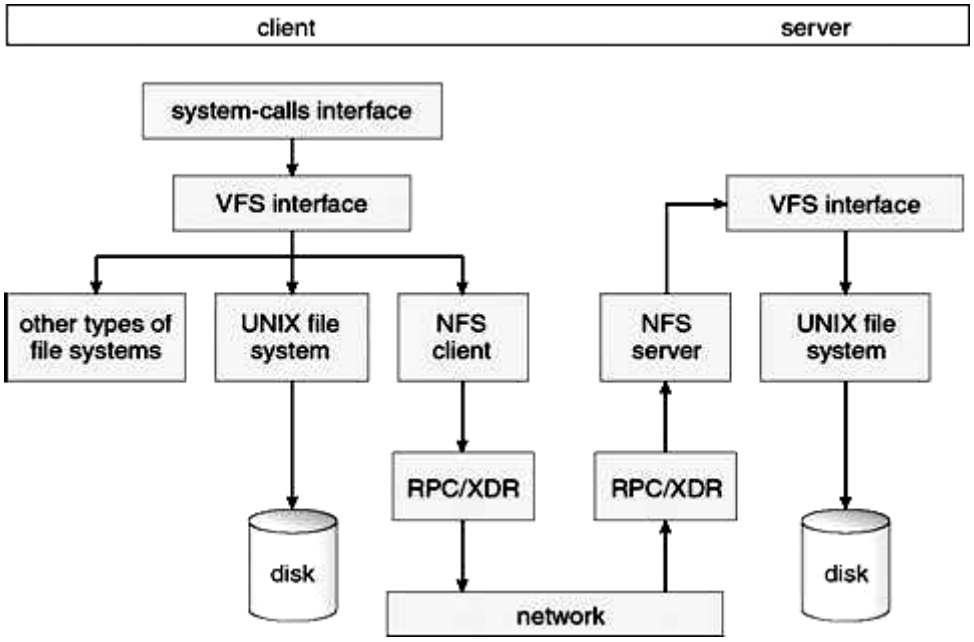


Fig 12.7 Schematic View of NFS Architecture

12.7 LET US SUM UP

- Long-term existence: Files are stored on disk or other secondary storage and do not disappear when a user logs off.
- Sharable between processes: Files have names and can have associated access permissions that permit controlled sharing.
- A field is the basic element of data. An individual field contains a single value.
- A record is a collection of related fields that can be treated as a unit by some application program.
- A file is a collection of similar records.
- A database is a collection of related data.
- The first reference to a file (normally an open) causes the directory structure to be searched and the directory entry for this file to be copied into the table of opened files

- The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk
- First Fit and Best Fit are the most common strategies used to select a free hole from the set of available holes.
- External fragmentation exists whenever free space is broken into chunks.
- The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file.
- **Network File Systems [NFS]** are standard UNIX client-server file sharing protocol.
- NFS servers are stateless; each request has to provide a full set of arguments.

12.8 UNIT END QUESTIONS

1. Define the terms :
    a. Field
    b. Record
    c. Database
2. What are the objectives for a file management system?
3. What is File System Mounting? Explain Contiguous Allocation method in detail.
4. What is the difference between Linked Allocation and Indexed Allocation methods?
5. Write a short note Bit Vector.
6. What is NFS? What are its protocols?

13

MASS-STORAGE STRUCTURE AND DEADLOCKS

Unit Structure

13.0 OBJECTIVES

After reading this unit you will be able to:

- Describe boot blocks and bad blocks
- Distinguish between various levels of RAID
- Define deadlock
- Detect, recover, avoid and prevent deadlocks

13.1 INTRODUCTION

In order to get vast amount of storage capacity of several bytes (trillions and more) in a computer system, a different kind of storage system is used. In such type of system, multiple units of similar kinds of storage media are associated together to form a chain of mass storage devices. These storage media may include multiple magnetic tape reels or cartridges, multiple arrays of magnetic disks or multiple CD-ROMs as a storage device. We can categorize mass storage devices into three types:

1. Redundant Array of Inexpensive Disks (RAID)
2. Automated Tape Library
3. CD-ROM Jukebox

13.2 DISK STRUCTURE

The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to have a different logical block size. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder.

In practice, it is difficult to perform this translation, for two reasons. First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives.

On media that uses constant linear velocity (CLV), the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as constant angular velocity (CAV).

The number of sectors per track has been increasing now-a-days and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

13.3 DISK MANAGEMENT

13.3.1 DISK FORMATTING

Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting or physical formatting. Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC).

The ECC contains enough information, if only a few bits of data have been corrupted, to enable the controller to identify which bits have changed and calculate what their correct values should be. It then reports a recoverable soft error. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk.

The second step is logical formatting, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or inodes) and an initial empty directory.

13.3.1 BOOT BLOCK

The initial bootstrap (stored in ROM for most computers) program initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system by finding the OS Kernel on disk, loading it in memory and jumping to an initial address to begin the OS execution.

The full bootstrap program is stored in the "boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk. Several major kinds of boot sectors could be encountered on IBM PC compatible hard disks, floppy disks and similar storage devices:

- **A Master Boot Record (MBR)** is the first sector of a data storage device that has been partitioned. The MBR sector may contain code to locate the active partition and invoke its Volume Boot Record.

- **A Volume Boot Record (VBR)** is the first sector of a data storage device that has not been partitioned, or the first sector of an individual partition on a data storage device that has been partitioned. It may contain code to load and invoke an operating system (or other standalone program) installed on that device or within that partition.

13.3.1 BAD BLOCKS

A bad sector is a sector on a computer's disk drive or flash memory that cannot be used due to permanent damage (or an OS inability to successfully access it), such as physical damage to the disk surface (or sometimes sectors being stuck in a magnetic or digital state that cannot be reversed) or failed flash memory transistors. It is usually detected by a disk utility software such as CHKDSK or SCANDISK on Microsoft systems, or badblocks on Unix-like systems. When found, these programs may mark the sectors unusable (all file systems contain provisions for bad-sector marks) and the operating system skips them in the future.

More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector-sparing or forwarding.

As an alternative to sector sparing, some controllers can be instructed to replace a bad block by sector slipping. During a low-level format, defect lists are populated which store which sectors are bad. The bad sectors are then mapped and a sector slipping algorithm is utilized. When using sector slipping for bad sectors, disk access time is not largely affected. The drive will just skip over the sectors and takes the same time as it would have to read the sector. Spare sectors are located on the disk to aid in having sectors to "slip" other sectors down to, allowing for the preservation of sequential ordering of the data. Accuracy of programs reliant on static knowledge of cylinders and block positions will be compromised though.

13.4 SWAP SPACE MANAGEMENT

Swap-space management refers to the process where OS breaks the physical RAM that is structured with random access memory into some pitches of memory called pages. This is the method in which a single page of memory can be copied to the

preconfigured space on a hard disk. This is called swap space. This process helps to free up that page of memory. An assessment help can say that these sizes when considered altogether of the physical memory as well as the swap space can be recognized in terms of total virtual memory that is available for the whole system.

13.4.1 SWAP SPACE USE

Systems that implement swapping may use swap space to hold an entire process image including the code and data segments. Paging systems may simply store pages that have been pushed out of main memory. The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes depending on the amow1.t of physical memory the amount of virtual memory it is backing and the way in which the virtual memory is used.

13.4.2 SWAP SPACE LOCATION

A swap space can reside in one of two places: it can be carved out of the normal file system, or it can be in a separate disk partition. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it and allocate its space. Alternatively, swap space can be created in a separate partition. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and de-allocate the blocks from the raw partition. This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems.

13.5 RAID STRUCTURE

A variety of disk-organization techniques, collectively called *redundant arrays of inexpensive disks* (RAIDs) are commonly used to address the performance and reliability issues. In the past, RAIDs composed of small, cheap disks were viewed as a cost-effective alternative to large, expensive disks. Today, RAIDs are used for their higher reliability and higher data-transfer rate, rather than for economic reasons. Hence, the I in RAID now stands for ‘‘independent” instead of “inexpensive”.

13.5.1 Improvement of Reliability via Redundancy

Suppose that the mean time to failure of a single disk is 100000 hours. Then the mean time to failure of some disk in an array of 100 disks will be 100000/100 = 1000 hours, or 41.66 days, which is not long at all. If we store only one copy of the data, then each disk failure will result in loss of a significant amount of data-and such a high rate of data loss is unacceptable.

The solution to the problem of reliability is to introduce redundancy; we store extra information that is not normally needed but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost. The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring**.

13.5.2 Improvement in Performance via Parallelism

With multiple disks, we can improve the transfer rate as well (or instead) by striping data across the disks. In its simplest form, data striping consists of splitting the bits of each byte across multiple disks; such striping is called bit-level striping. For example, if we have an array of eight disks, we write bit i of each byte to disk i. The array of eight disks can be treated as a single disk with sectors that are eight times the normal size and, more important that have eight times the access rate. Parallelism in a disk system, as achieved through striping, has two main goals:

1. Increase the throughput of multiple small accesses (that is, page accesses) by load balancing.
2. Reduce the response time of large accesses.

13.5.3 RAID Levels

- Mirroring provides high reliability, but it is expensive.
- Striping provides high data-transfer rates, but it does not improve reliability.
- Numerous schemes to provide redundancy at lower cost by using the idea of disk striping combined with ‘‘parity” bits have been proposed.
- These schemes have different cost-performance trade-offs and are classified according to levels called **RAID levels**
- (in the figure, P indicates error-correcting bits, and C indicates a second copy of the data).
- In all cases depicted in the figure, four disks’ worth of data are stored, and the extra disks are used to store redundant information for failure recovery.

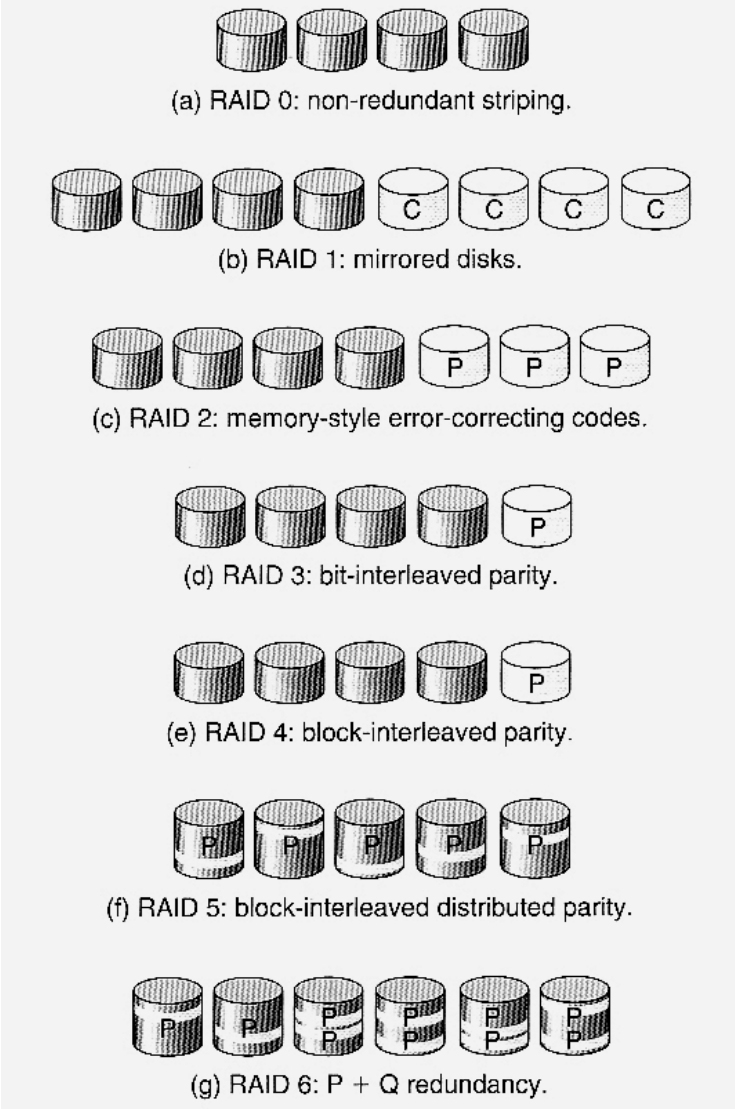


Fig 13.1 RAID Levels

- **RAID Level 0**. RAID level 0 refers to disk arrays with striping at the level of blocks but without any redundancy (such as mirroring or parity bits).
- **RAID Level 1**. RAID level 1 refers to disk mirroring.
- **RAID Level 2**. RAID level 2 is also known as **memory-style error-correcting-code (ECC) organization**. Memory systems have long detected certain errors by using parity bits.
- **RAID Level 3**. RAID level 3, or **bit-interleaved parity organization**, improves on level 2 by taking into account the fact that, unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction as well as for detection.
- **RAID Level 4**. RAID level 4, or block-interleaved parity organization, uses block-level striping, as in RAID 0, and in

addition keeps a parity block on a separate disk for corresponding blocks from N other disks.

- **RAID Level 5**. RAID level 5, or block-interleaved distributed parity, differs from level 4 by spreading data and parity among all *N+1* disks, rather than storing data in N disks and parity in one disk. For each block, one of the disks stores the parity, and the others store data.
- **RAID Level 6**. RAID level 6, also called the *P+Q* redundancy scheme, is much like RAID level 5 but stores extra redundant information to guard against multiple disk failures.
- **RAID Level 0 + 1**. RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, this level provides better performance than RAID 5. It is common in environments where both performance and reliability are important. Unfortunately, it doubles the number of disks needed for storage, as does RAID I, so it is also more expensive.

13.5.4 Selecting a RAID Level

Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.

Other decisions include how many disks are involved in a RAID set and how many disks to protect with a single parity bit. More disks in the set increases performance but increases cost. Protecting more disks per parity bit saves cost, but increases the likelihood that a second disk will fail before the first bad disk is repaired.

13.5.5 Extensions

RAID concepts have been extended to tape drives (e.g. striping tapes for faster backups or parity checking tapes for reliability), and for broadcasting of data.

13.5.6 Problems with RAID

RAID protects against physical errors, but not against any number of bugs or other errors that could write erroneous data. ZFS (Z file system, developed by Sun™, is a new technology designed to use a pooled storage method) adds an extra level of protection by including data block checksums in all inodes along with the pointers to the data blocks. If data are mirrored and one copy has the correct checksum and the other does not, then the data with the bad checksum will be replaced with a copy of the data with the good checksum. This increases reliability greatly over

RAID alone, at a cost of a performance hit that is acceptable because ZFS is so fast to begin with.

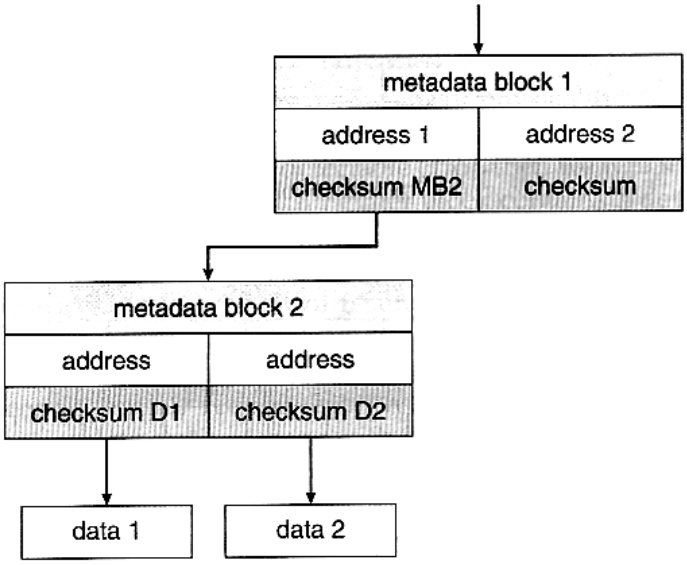


Fig 13.2 ZFS checksums all metadata and data

Another problem with traditional file systems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust file system sizes, because a file system cannot span across multiple file systems.

ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to file systems as needed. File system sizes can be limited by quotas, and space can also be reserved to guarantee that a file system will be able to grow later, but these parameters can be changed at any time by the file system's owner. Otherwise file systems grow and shrink dynamically as needed.

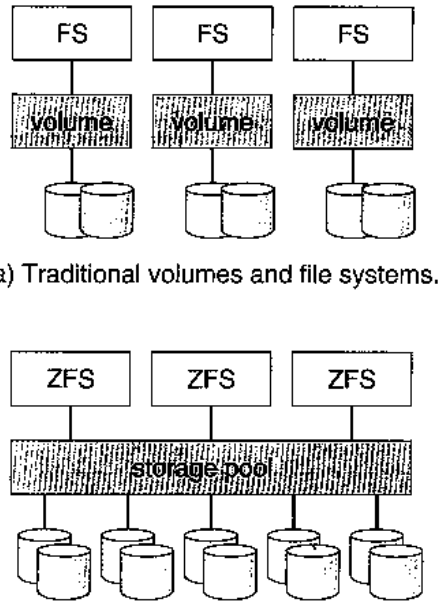


Fig 13.3 Traditional volumes and ZFS storage

13.6 STABLE STORAGE IMPLEMENTATION

The concept of stable storage involves a storage medium in which data is never lost, even in the face of equipment failure in the middle of a write operation. To implement this requires two (or more) copies of the data, with separate failure modes. An attempted disk write results in one of three possible outcomes:

- The data is successfully and completely written.
- The data is partially written, but not completely. The last block written may be garbled.
- No writing takes place at all.

Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state. To do this requires two physical blocks for every logical block, and the following procedure:

- Write the data to the first physical block.
- After step 1 had completed, then write the data to the second physical block.
- Declare the operation complete only after both physical writes have completed successfully.

During recovery the pair of blocks is examined.

- If both blocks are identical and there is no sign of damage, then no further action is necessary.
- If one block contains a detectable error but the other does not, then the damaged block is replaced with the good copy. (This will either undo the operation or complete the operation, depending on which block is damaged and which is undamaged)
- If neither block shows damage but the data in the blocks differ, then replace the data in the first block with the data in the second block. (Undo the operation)

Because the sequence of operations described above is slow, stable storage usually includes NVRAM as a cache, and declares a write operation complete once it has been written to the NVRAM.

13.7 DEADLOCKS

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes.

If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly. A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.

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

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

Deadlock Characterization

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

Necessary Conditions

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

1. Mutual exclusion: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the

resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

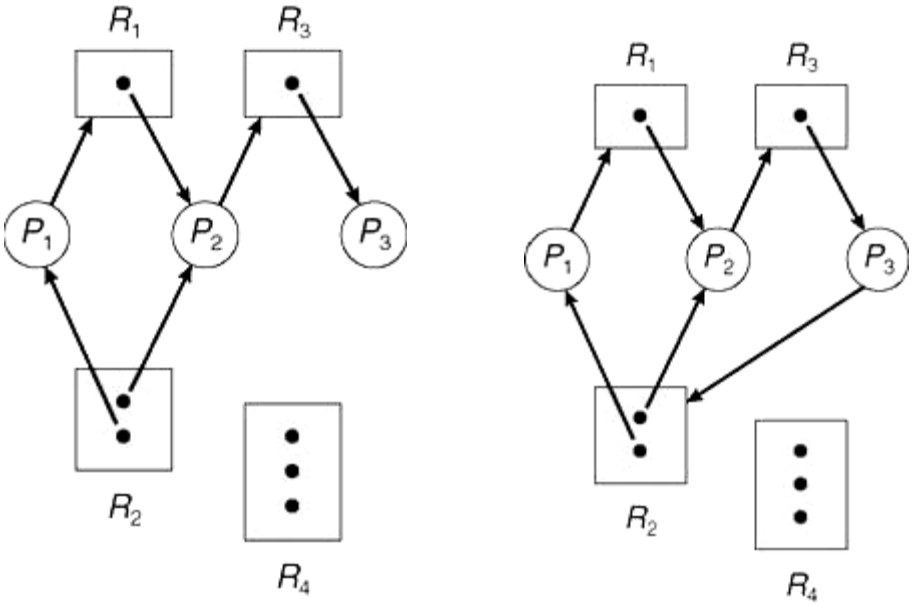
2. Hold and wait: There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

3. No pre-emption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process, has completed its task.

4. Circular wait: There must exist a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P_2, —, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. The set of vertices V is partitioned into two different types of nodes $P = \{P_0, P_1, ..., P_n\}$ the set consisting of all the active processes in the system; and $R = \{R_0, R_1, ..., R_n\}$, the set consisting of all resource types in the system.



(a) (b)

Fig 13.4 (a) Example of Resource allocation graph (b) Resource allocation graph with a deadlock

A directed edge from process P_i to resource type R_j, is denoted by $P_i \rightarrow R_j$, it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by

$R_j \rightarrow P_i$. It signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \rightarrow R_j$ is called a request edge; a directed edge $R_j \rightarrow P_i$ is called an assignment edge.

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

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

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

Methods for Handling Deadlocks

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

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

Deadlock prevention is a set of methods for ensuring that at least one of the necessary cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance requires the operating system to have, in advance, additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each request whether or not the process should wait. Each request requires that the system considers the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed.

13.1 Deadlock Prevention

For a deadlock to occur, each of the four necessary-conditions must hold.

1. Mutual Exclusion

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

2. Hold and Wait

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

2. An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated.

There are two main disadvantages to these protocols. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file.

If we cannot be assured that they will, then we must request all resources at the beginning for both protocols. Second, starvation is possible.

3. No Preemption

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

4. Circular Wait

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

Let $R = \{R_1, R_2, ..., R_n\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows to compare two resources and to determine whether one

precedes another in ordering. Formally, we define a one-to-one function F: R→N, where N is the set of natural numbers.

13.2 Deadlock Avoidance

Prevent deadlocks requests can be made. The restraints ensure that at least one of the necessary conditions for deadlock to not occur, and, hence, those deadlocks cannot hold. Possible side effects of preventing deadlocks by this, melted, however, are Tow device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, we might be told that process P will request the tape drive first and later the printer, before releasing both resources. Process Q on the other hand, will request the printer first and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process we can decide for each request whether or not the process should wait.

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure there can never be a circular wait condition. The resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

1. Safe State

A state is safe if the system allocates resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there of processes $< P_1, P_2, ..., P_n >$ is in a safe sequence for the current allocation state if, for each P_i the resources that P_j can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j<i$. In this situation, if the resources that process P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

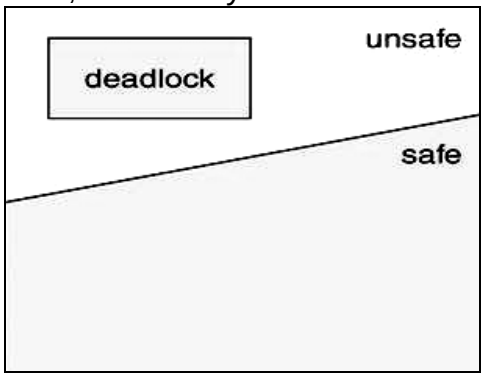
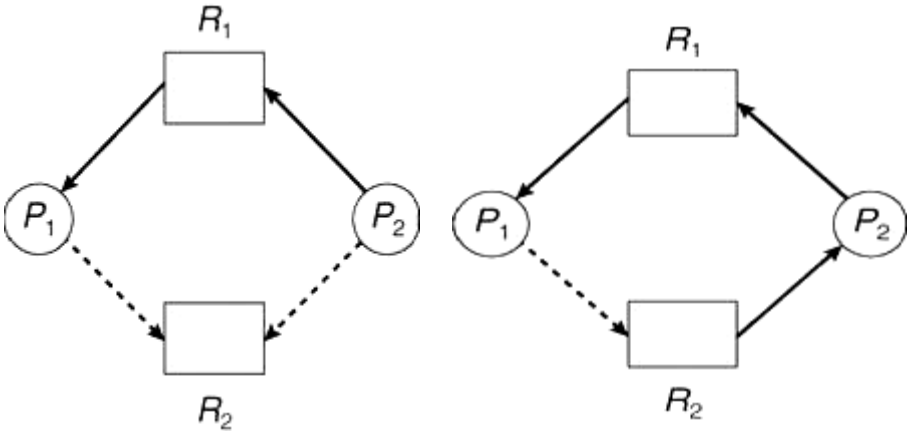


Fig 13.5 Safe, Unsafe deadlock state

2. Resource-Allocation Graph Algorithm

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



(a) **(b)**

Fig 13.6 (a) Resource-Allocation Graph
(b) Unsafe state in Resource-Allocation Graph

3. Banker's Algorithm

The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances of each resource type. The deadlock-avoidance algorithm that we describe next is applicable to such a system, but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm.

13.7.3 Deadlock Detection

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

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

13.7.3.1 Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.

13.7.3.2 Several Instances of a Resource Type

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

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

13.7.3.3 Detection-Algorithm Usage

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken.

13.7.4 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has spurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically.

There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

13.7.4.1 Process Termination

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

- **Abort all deadlocked processes:** This method clearly will break the dead - lock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed

- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted a deadlock-detection algorithm must be invoked to determine whether a processes are still deadlocked.

13.7.4.2 Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give

these resources to other processes until he deadlock cycle is broken. The three issues are considered to recover from deadlock

1. Selecting a victim (to minimize cost)
2. Rollback (return to some safe state, restart process for that state)
3. Starvation (same process may always be picked as victim, include number of rollback in cost factor)

13.8 LET US SUM UP

- On media that uses constant linear velocity (CLV), the density of bits per track is uniform
- The ECC contains enough information, if only a few bits of data have been corrupted, to enable the controller to identify which bits have changed and calculate what their correct values should b
- **A Master Boot Record (MBR)** is the first sector of a data storage device that has been partitioned.
- **A Volume Boot Record (VBR)** is the first sector of a data storage device that has not been partitioned, or the first sector of an individual partition on a data storage device that has been partitioned.
- A bad sector is a sector on a computer's disk drive or flash memory that cannot be used due to permanent damage (or an OS inability to successfully access it), such as physical damage to the disk surface (or sometimes sectors being stuck in a magnetic or digital state that cannot be reversed) or failed flash memory transistors.
- Swap-space management refers to the process where OS breaks the physical RAM that is structured with random access memory into some pitches of memory called pages.
- A variety of disk-organization techniques, collectively called *redundant arrays of inexpensive disks* (RAIDs) are commonly used to address the performance and reliability issues.
- The solution to the problem of reliability is to introduce redundancy; we store extra information that is not normally needed but that can be used in the event of failure of a disk to rebuild the lost information
- Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state
- In deadlock, processes never finish executing and system resources are tied up, preventing other jobs from ever starting

- If each resource type has several instances, then a cycle implies that a deadlock has occurred.
- Deadlock prevention is a set of methods for ensuring that at least one of the necessary cannot hold
- Deadlock avoidance requires the operating system to have, in advance, additional information concerning which resources a process will request and use during its lifetime

13.9 UNIT END QUESTIONS

1. Define boot block and bad blocks.
2. Describe various RAID levels with the help of diagrams.
3. What are deadlocks? What are the necessary conditions arised in a deadlock situation?
4. How to:
   a) Detect a deadlock
   b) Recover from a deadlock
5. Define
   a. Low-level formatting
   b. Swap Space
   c. RAID
   d. Mirroring

14

I/O SYSTEMS

Unit Structure

14.0 OBJECTIVES

After reading this unit you will be able to:

- Discuss structuring techniques and interfaces for the operating system that enable I/0 devices to be treated in a standard, uniform way.
- Explain how an application can open a file on a disk without knowing what kind of disk it is and how new disks and other devices can be added to a computer without disruption of the operating system

14.1 INTRODUCTION

Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals)

I/O Subsystems must contend with two trends:

(1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and

(2) The development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.

Device drivers are modules that can be plugged into an OS to handle a particular device or category of similar devices.

14.2 APPLICATION I/O INTERFACE

I/O system calls encapsulate device behaviors in generic classes. Device-driver layer hides differences among I/O controllers 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

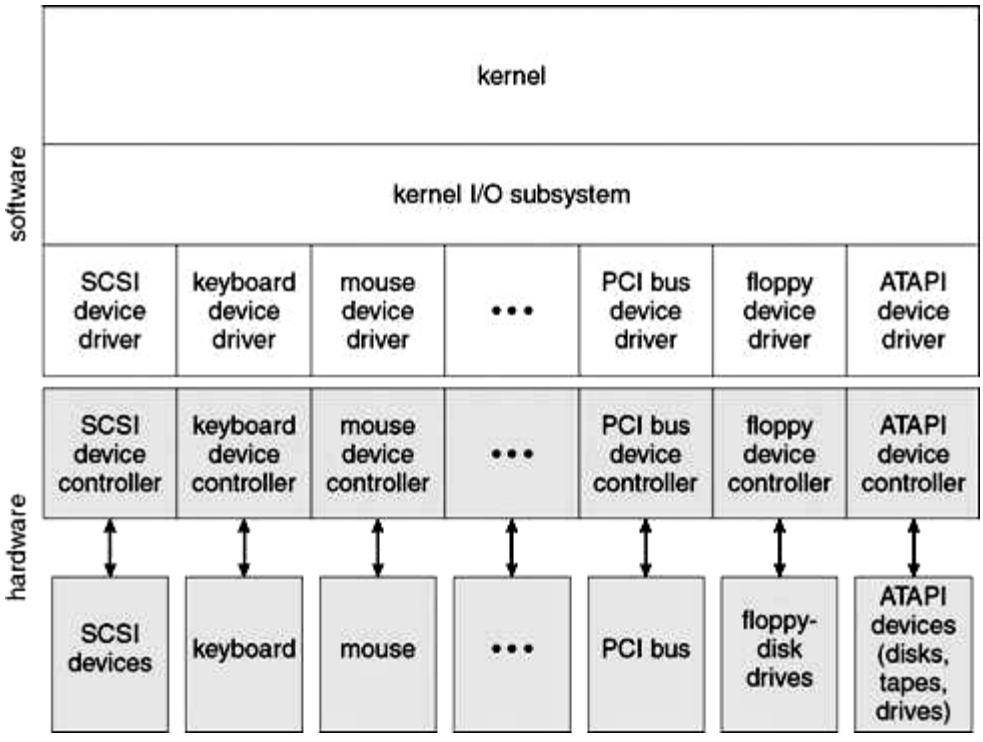


Fig 14.1 A Kernel I/O Structure

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Fig 14.2 Characteristics of I/O devices

14.2.1 BLOCK AND CHARACTER DEVICES

Block devices are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include read(), write(), and seek().

Accessing blocks on a hard drive directly (without going through the filesystem structure) is called raw I/O, and can speed up certain operations by bypassing the buffering and locking normally conducted by the OS. (It then becomes the application's responsibility to manage those issues)

A new alternative is direct I/O, which uses the normal filesystem access, but which disables buffering and locking operations.

Memory-mapped file I/O can be layered on top of block-device drivers. Rather than reading in the entire file, it is mapped to a range of memory addresses, and then paged into memory as needed using the virtual memory system.

Access to the file is then accomplished through normal memory accesses, rather than through read() and write() system calls. This approach is commonly used for executable program code.

Character devices are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include get() and put(), with more advanced functionality such as reading an entire line supported by higher-level library routines.

14.2.2 NETWORK DEVICES

Because network access is inherently different from local disk access, most systems provide a separate interface for network devices. One common and popular interface is the socket interface, which acts like a cable or pipeline connecting two networked entities. Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer. The select() system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

14.2.3 CLOCKS AND TIMERS

Three types of time services are commonly needed in modern systems:

- Get the current time of day.
- Get the elapsed time (system or wall clock) since a previous event.
- Set a timer to trigger event X at time T.

Unfortunately time operations are not standard across all systems. A programmable interrupt timer, PIT can be used to trigger operations and to measure elapsed time. It can be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.

- The scheduler uses a PIT to trigger interrupts for ending time slices.
- The disk system may use a PIT to schedule periodic maintenance cleanup, such as flushing buffers to disk.
- Networks use PIT to abort or repeat operations that are taking too long to complete i.e. resending packets if an acknowledgement is not received before the timer goes off.
- More timers than actually exist can be simulated by maintaining an ordered list of timer events, and setting the physical timer to go off when the next scheduled event should occur.

On most systems the system clock is implemented by counting interrupts generated by the PIT. Unfortunately this is limited in its resolution to the interrupt frequency of the PIT, and may be subject to some drift over time. An alternate approach is to provide direct access to a high frequency hardware counter, which provides much higher resolution and accuracy, but which does not support interrupts.

14.2.4 BLOCKING AND NON-BLOCKING I/O

With blocking I/O a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime.

With non-blocking I/O the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the process to check for available data without getting hung completely if it is not there.

One approach for programmers to implement non-blocking I/O is to have a multi-threaded application, in which one thread makes blocking I/O calls (say to read a keyboard or mouse), while other threads continue to update the screen or perform other tasks.

A subtle variation of the non-blocking I/O is the asynchronous I/O, in which the I/O request returns immediately allowing the process to continue on with other tasks, and then the process is notified (via changing a process variable, or a software interrupt, or a callback function) when the I/O operation has completed and the data is available for use. (The regular non-blocking I/O returns immediately with whatever results are available, but does not complete the operation and notify the process later)

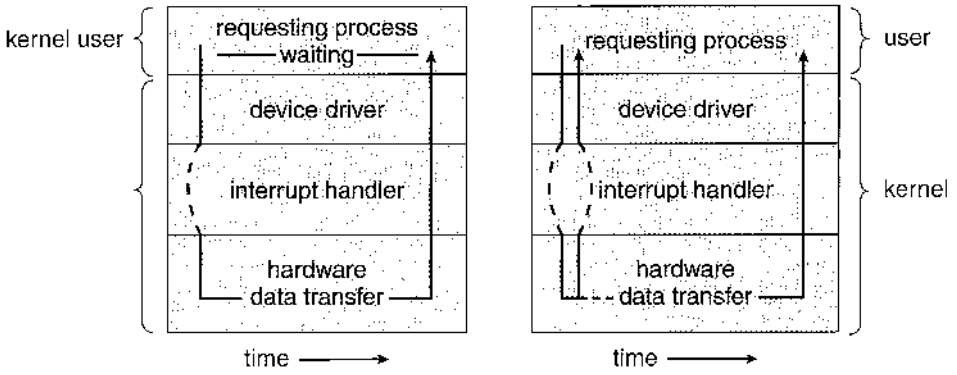


Fig14.3 Two I/O methods: (a) synchronous and (b) asynchronous

14.3 TRANSFORMING I/O REQUESTS TO HARDWARE OPERATIONS

Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.

DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.). UNIX uses a mount table to map filename prefixes (e.g. /usr) to specific mounted devices. Where multiple entries in the mount table match different prefixes of the filename

the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file)

UNIX uses special device files, usually located in /dev, to represent and access physical devices directly.

Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go. The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler).

The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver. (e.g. a particular disk drive or partition).

A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users. Figure 14.4 illustrates the steps taken to process a (blocking) read request:

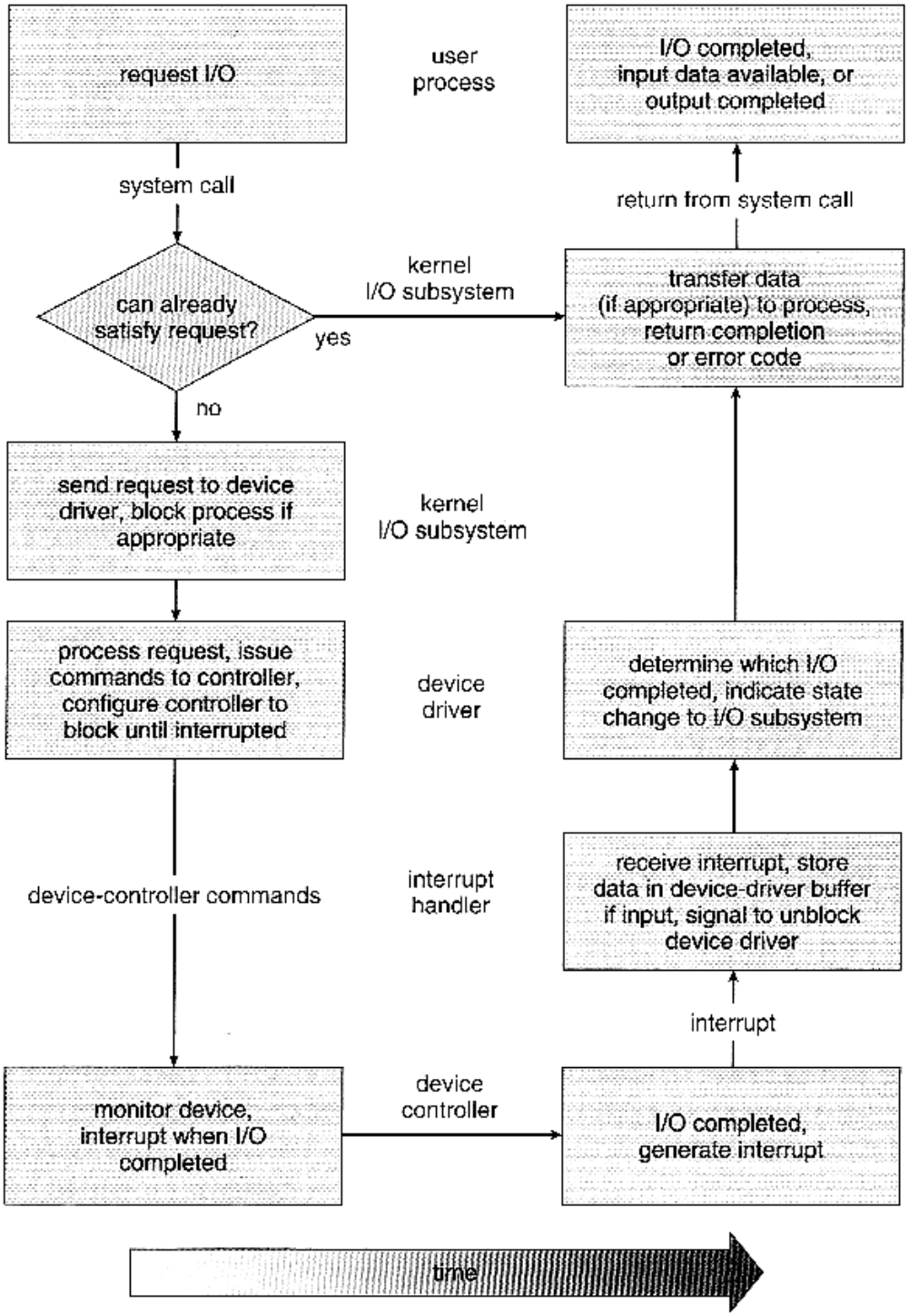


Fig 14.4 The life cycle of an I/O request

14.4 STREAMS

The streams mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added.

The user process interacts with the stream head. The device driver interacts with the device end. Zero or more stream modules can be pushed onto the stream, using ioctl(). These modules may filter and/or modify the data as it passes through the stream.

Each module has a read queue and a write queue. Flow control can be optionally supported, in which case each module will buffer data until the adjacent module is ready to receive it. Without flow control, data is passed along as soon as it is ready.

User processes communicate with the stream head using either read() and write() [or putmsg() and getmsg() for message passing]. Streams I/O is asynchronous (non-blocking), except for the interface between the user process and the stream head.

The device driver must respond to interrupts from its device - If the adjacent module is not prepared to accept data and the device driver's buffers are all full, then data is typically dropped.

Streams are widely used in UNIX, and are the preferred approach for device drivers. For example, UNIX implements sockets using streams.

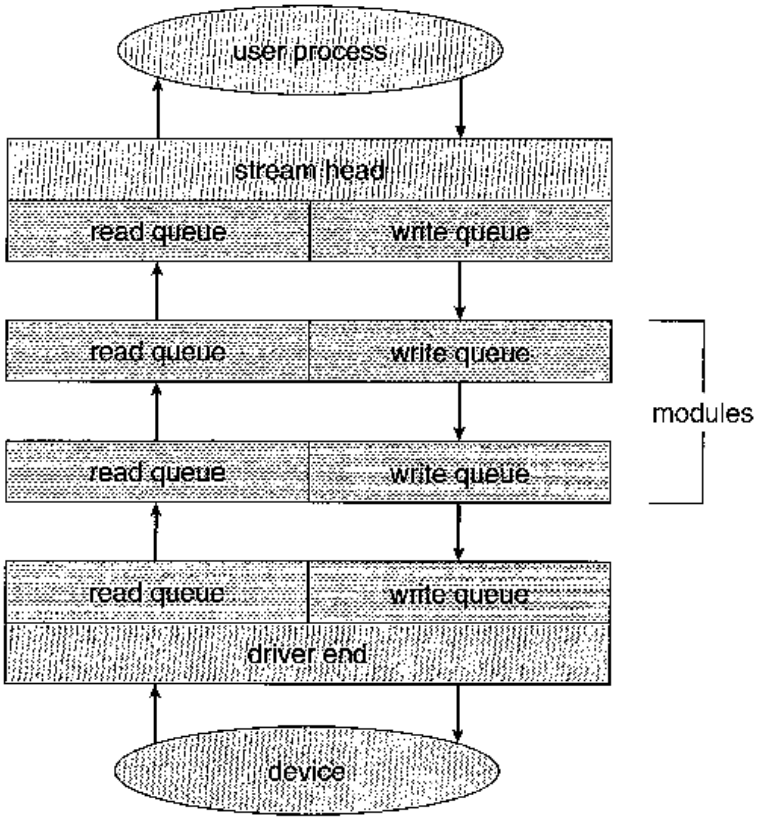


Fig 14.5 The STREAMS architecture

14.5 PERFORMANCE

The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few)

Interrupt handling can be relatively expensive (slow), which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive.

Network traffic can also put a heavy load on the system. Consider for example the sequence of events that occur when a single character is typed in a telnet session, as shown in figure 14.6. (And the fact that a similar set of events must happen in reverse to echo back the character that was typed) Sun uses in-kernel threads for the telnet daemon, increasing the supportable number of simultaneous telnet sessions from the hundreds to the thousands.

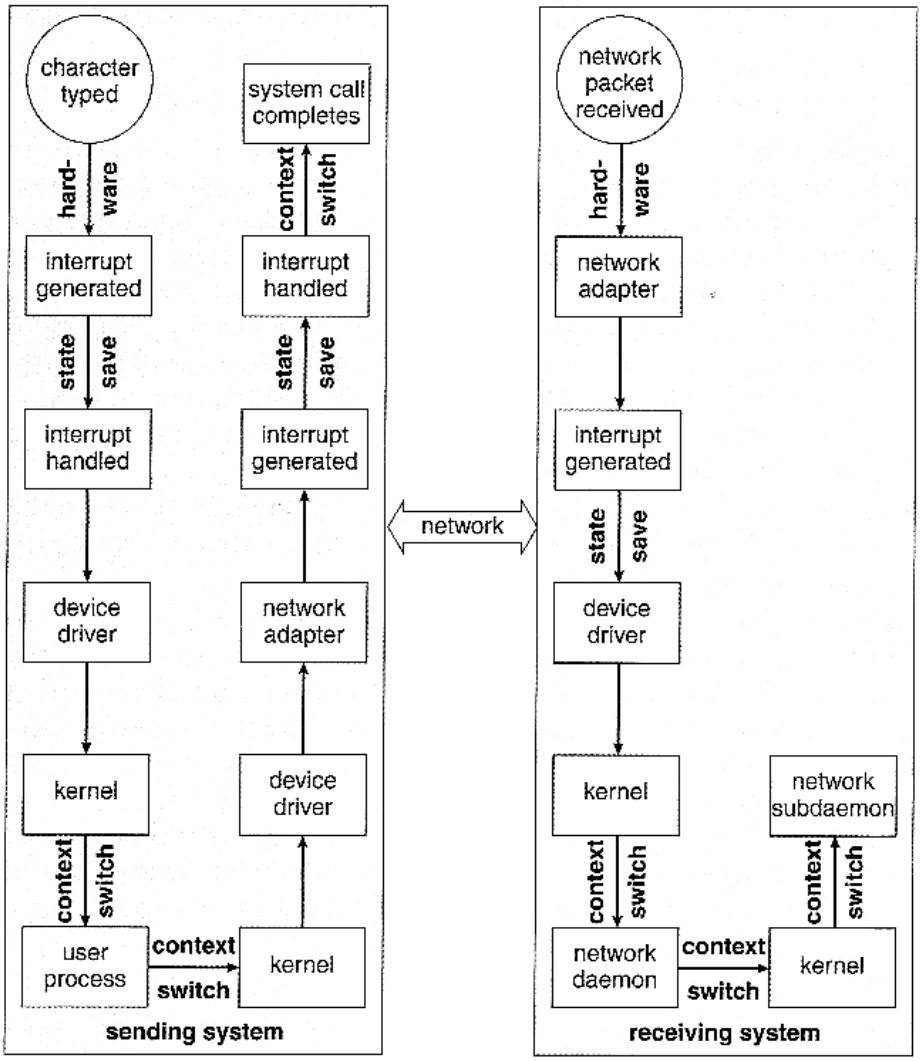


Fig 14.6 Intercomputer Communications

Other systems use front-end processors to off-load some of the work of I/O processing from the CPU. For example a terminal concentrator can multiplex with hundreds of terminals on a single port on a large computer.

Several principles can be employed to increase the overall efficiency of I/O processing:

- Reduce the number of context switches.
- Reduce the number of times data must be copied.
- Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.
- Increase concurrency using DMA.
- Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.

- Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.

The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation, as shown in Figure 14.7. Lower-level implementations are faster and more efficient, but higher-level ones are more flexible and easier to modify. Hardware-level functionality may also be harder for higher-level authorities (e.g. the kernel) to control.

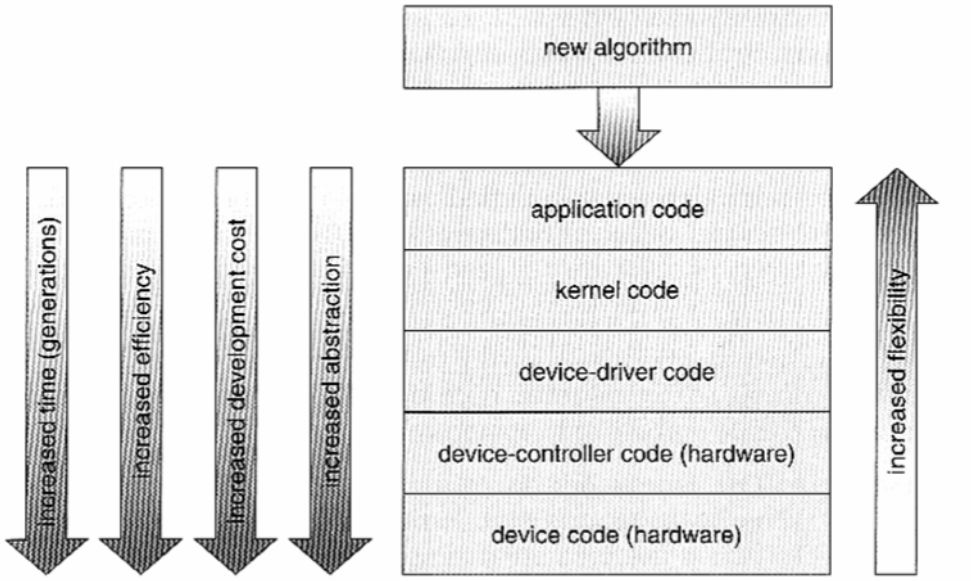


Fig 14.7 Device Functionality Progression

14.6 LET US SUM UP

- 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 devices are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems
- The select() system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.
- One approach for programmers to implement non-blocking I/O is to have a multi-threaded application, in which one thread makes blocking I/O calls (say to read a keyboard or mouse),

while other threads continue to update the screen or perform other tasks.

- Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.
- The streams mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added.
- Streams I/O is asynchronous (non-blocking), except for the interface between the user process and the stream head.
- The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation.

14.7 UNIT END QUESTIONS

7. State some characteristics of I/O devices.
8. Write a short note on Blocking and non-blocking I/O
9. Describe the life cycle of an I/O request with the help of a diagram.
10. What are STREAMS?
11. State various principles employed to increase the overall efficiency of I/O processing.

15

PROTECTION AND SECURITY

Unit Structure

15.0 OBJECTIVES

After studying this unit, you will be able:

- To ensure that each shared resource is used only in accordance with system policies, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the mechanisms for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

15.1 INTRODUCTION

The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system. Still others are defined by the individual users to protect their own files and programs. A protection system must have the flexibility to enforce a variety of policies.

15.2 PRINCIPLES OF PROTECTION

The principle of least privilege dictates that programs, users, and systems be given just enough privileges to perform their tasks. This ensures that failures do the least amount of harm and allow the least of harm to be done.

For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.

Typically each user is given their own account, and has only enough privilege to modify their own files. The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

15.3 DOMAIN OF PROTECTION

A computer can be viewed as a collection of processes and objects (both hardware and software).

The need to know principle states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access. The modes available for a particular object may depend upon its type.

15.3.1 Domain Structure

A protection domain specifies the resources that a process may access. Each domain defines a set of objects and the types of operations that may be invoked on each object. An access right is the ability to execute an operation on an object.

A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.

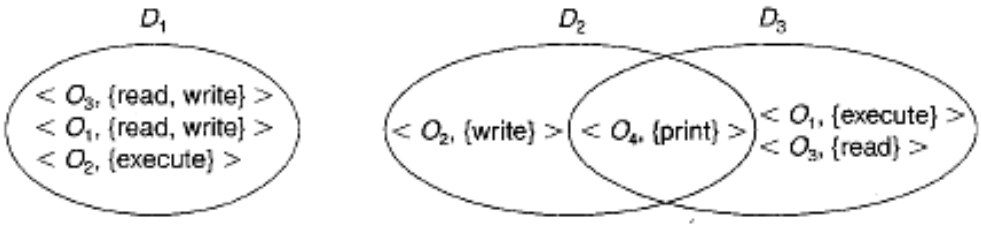


Fig 15.1 System with three protection domains

The association between a process and a domain may be static or dynamic. If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically. If the association is dynamic, then there needs to be a mechanism for domain switching.

Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID.

15.3.2 An Example: UNIX

UNIX associates domains with users. Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running (and similarly for the SGID bit). Unfortunately this has some potential for abuse.

An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.

Yet another alternative is to not allow the changing of ID at all. Instead, special privileged daemons are launched at boot time, and

user processes send messages to these daemons when they need special tasks performed.

15.4 ACCESS MATRIX

The model of protection that we have been discussing can be viewed as an access matrix, in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Fig 15.2 Access Matrix

Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Fig 15.3 Access matrix of fig.15.2 with domains as objects

The ability to copy rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:

If the asterisk is removed from the original access right, then the right is transferred, rather than being copied. This may be termed a transfer right as opposed to a copy right.

If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a limited copy right, as shown in Figure 15.4 below:

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Fig 15.4 Access matrix with copyrights

The owner right adds the privilege of adding new rights or removing existing ones:

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Fig 15.5 Access matrix with owner rights

Copy and owner rights only allow the modification of rights within a column. The addition of control rights, which only apply to domain objects, allow a process operating in one domain to affect the rights available in other domains. For example in the table below, a process operating in domain D2 has the right to control any of the rights in domain D4.

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Fig 15.6 Modified access matrix of fig. 15.3

15.5 ACCESS CONTROL

Role-Based Access Control, RBAC, assigns privileges to users, programs, or roles as appropriate, where "privileges" refer to the right to call certain system calls, or to use certain parameters with those calls. RBAC supports the principle of least privilege, and reduces the susceptibility to abuse as opposed to SUID or SGID programs.

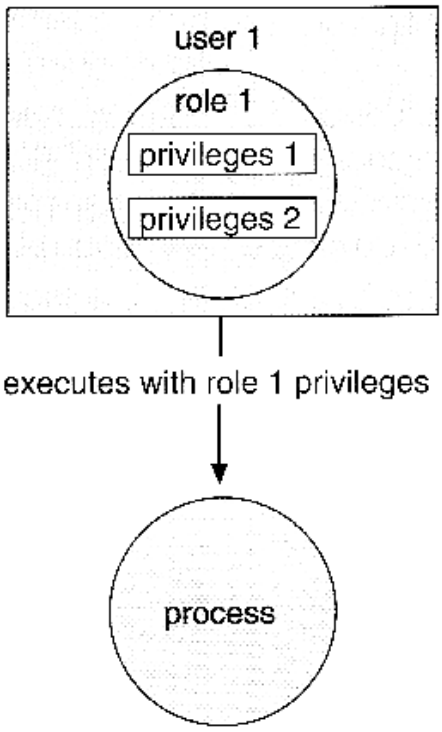


Fig 15.7 Role-based access control in Solaris 10

15.6 CAPABILITY-BASED SYSTEMS

15.6.1 An Example: Hydra

Hydra is a capability-based system that includes both system-defined rights and user-defined rights. The interpretation of user-defined rights is up to the specific user programs, but the OS provides support for protecting access to those rights, whatever they may be

Operations on objects are defined procedurally, and those procedures are themselves protected objects, accessed indirectly through capabilities. The names of user-defined procedures must

be identified to the protection system if it is to deal with user-defined rights.

When an object is created, the names of operations defined on that object become auxiliary rights, described in a capability for an instance of the type. For a process to act on an object, the capabilities it holds for that object must contain the name of the operation being invoked. This allows access to be controlled on an instance-by-instance and process-by-process basis.

Hydra also allows rights amplification, in which a process is deemed to be trustworthy, and thereby allowed to act on any object corresponding to its parameters.

Programmers can make direct use of the Hydra protection system, using suitable libraries which are documented in appropriate reference manuals.

15.6.2 An Example: Cambridge CAP System

The CAP system has two kinds of capabilities:

- Data capability, used to provide read, write, and execute access to objects. These capabilities are interpreted by microcode in the CAP machine.

- Software capability, is protected but not interpreted by the CAP microcode.

Software capabilities are interpreted by protected (privileged) procedures, possibly written by application programmers.

When a process executes a protected procedure, it temporarily gains the ability to read or write the contents of a software capability. This leaves the interpretation of the software capabilities up to the individual subsystems, and limits the potential damage that could be caused by a faulty privileged procedure.

Note, however, that protected procedures only get access to software capabilities for the subsystem of which they are a part. Checks are made when passing software capabilities to protected procedures that they are of the correct type.

Unfortunately the CAP system does not provide libraries, making it harder for an individual programmer to use than the Hydra system.

15.7 LANGUAGE-BASED PROTECTION

As systems have developed, protection systems have become more powerful, and also more specific and specialized. To refine

protection even further requires putting protection capabilities into the hands of individual programmers, so that protection policies can be implemented on the application level, i.e. to protect resources in ways that are known to the specific applications but not to the more general operating system.

15.7.1 Compiler-Based Enforcement

In a compiler-based approach to protection enforcement, programmers directly specify the protection needed for different resources at the time the resources are declared.

This approach has several advantages:

1. Protection needs are simply declared, as opposed to a complex series of procedure calls.
2. Protection requirements can be stated independently of the support provided by a particular OS.
3. The means of enforcement need not be provided directly by the developer.
4. Declarative notation is natural, because access privileges are closely related to the concept of data types.

Regardless of the means of implementation, compiler-based protection relies upon the underlying protection mechanisms provided by the underlying OS, such as the Cambridge CAP or Hydra systems.

Even if the underlying OS does not provide advanced protection mechanisms, the compiler can still offer some protection, such as treating memory accesses differently in code versus data segments. (E.g. code segments cant be modified, data segments can’t be executed)

There are several areas in which compiler-based protection can be compared to kernel-enforced protection:

Security

Security provided by the kernel offers better protection than that provided by a compiler. The security of the compiler-based enforcement is dependent upon the integrity of the compiler itself, as well as requiring that files not be modified after they are compiled. The kernel is in a better position to protect itself from modification, as well as protecting access to specific files. Where hardware support of individual memory accesses is available, the protection is stronger still.

Flexibility

A kernel-based protection system is not as flexible to provide the specific protection needed by an individual programmer, though

it may provide support which the programmer may make use of. Compilers are more easily changed and updated when necessary to change the protection services offered or their implementation.

Efficiency

The most efficient protection mechanism is one supported by hardware and microcode. Insofar as software based protection is concerned, compiler-based systems have the advantage that many checks can be made off-line, at compile time, rather that during execution.

The concept of incorporating protection mechanisms into programming languages is in its infancy, and still remains to be fully developed. However the general goal is to provide mechanisms for three functions:

1. Distributing capabilities safely and efficiently among customer processes. In particular a user process should only be able to access resources for which it was issued capabilities.
2. Specifying the type of operations a process may execute on a resource, such as reading or writing.
3. Specifying the order in which operations are performed on the resource, such as opening before reading.

15.7.2 Protection in Java

Java was designed from the very beginning to operate in a distributed environment, where code would be executed from a variety of trusted and untrusted sources. As a result the Java Virtual Machine, JVM incorporates many protection mechanisms

When a Java program runs, it load up classes dynamically, in response to requests to instantiates objects of particular types. These classes may come from a variety of different sources, some trusted and some not, which requires that the protection mechanism be implemented at the resolution of individual classes, something not supported by the basic operating system.

As each class is loaded, it is placed into a separate protection domain. The capabilities of each domain depend upon whether the source URL is trusted or not, the presence or absence of any digital signatures on the class, and a configurable policy file indicating which servers a particular user trusts, etc.

When a request is made to access a restricted resource in Java, (e.g. open a local file), some process on the current call stack must specifically assert a privilege to perform the operation. In essence this method assumes responsibility for the restricted access. Naturally the method must be part of a class which resides in a protection domain that includes the capability for the requested

operation. This approach is termed stack inspection, and works like this:

1. When a caller may not be trusted, a method executes an access request within a doPrivileged() block, which is noted on the calling stack.
2. When access to a protected resource is requested, check Permissions() inspects the call stack to see if a method has asserted the privilege to access the protected resource.
   - If a suitable doPriveleged block is encountered on the stack before a domain in which the privilege is disallowed, then the request is granted.
   - If a domain in which the request is disallowed is encountered first, then the access is denied and an AccessControlException is thrown.
   - If neither is encountered, then the response is implementation dependent.

In the example below the untrusted applet's call to get() succeeds, because the trusted URL loader asserts the privilege of opening the specific URL lucent.com. However when the applet tries to make a direct call to open() it fails, because it does not have privilege to access any sockets.

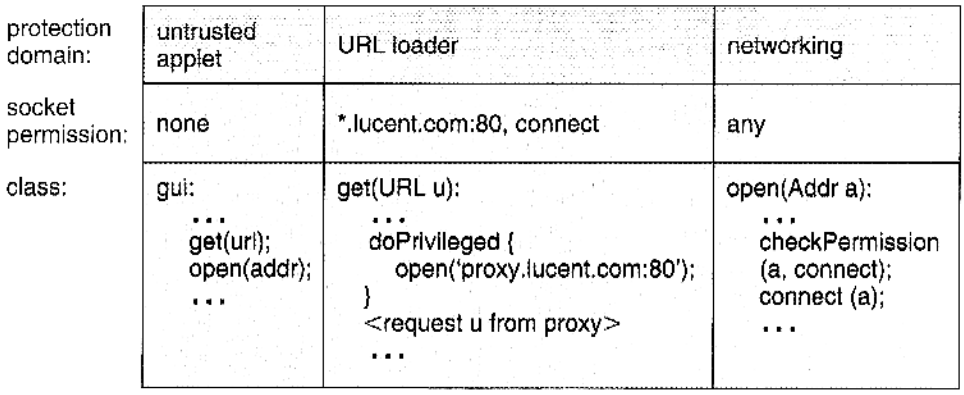


Fig 15.8 Stack Inspection

15.8 THE SECURITY PROBLEM

Some of the most common types of violations include:

Breach of Confidentiality - Theft of private or confidential information, such as credit-card numbers, trade secrets, patents, secret formulas, manufacturing procedures, medical information, financial information, etc.

Breach of Integrity - Unauthorized modification of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up

security holes on users systems before being released to the public.

Breach of Availability - Unauthorized destruction of data, often just for the "fun" of causing havoc and for bragging rites. Vandalism of web sites is a common form of this violation.

Theft of Service - Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services.

Denial of Service, DOS - Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service.

One common attack is masquerading, in which the attacker pretends to be a trusted third party. A variation of this is the man-in-the-middle, in which the attacker masquerades as both ends of the conversation to two targets.

A replay attack involves repeating a valid transmission. Sometimes this can be the entire attack, (such as repeating a request for a money transfer), or other times the content of the original message is replaced with malicious content.

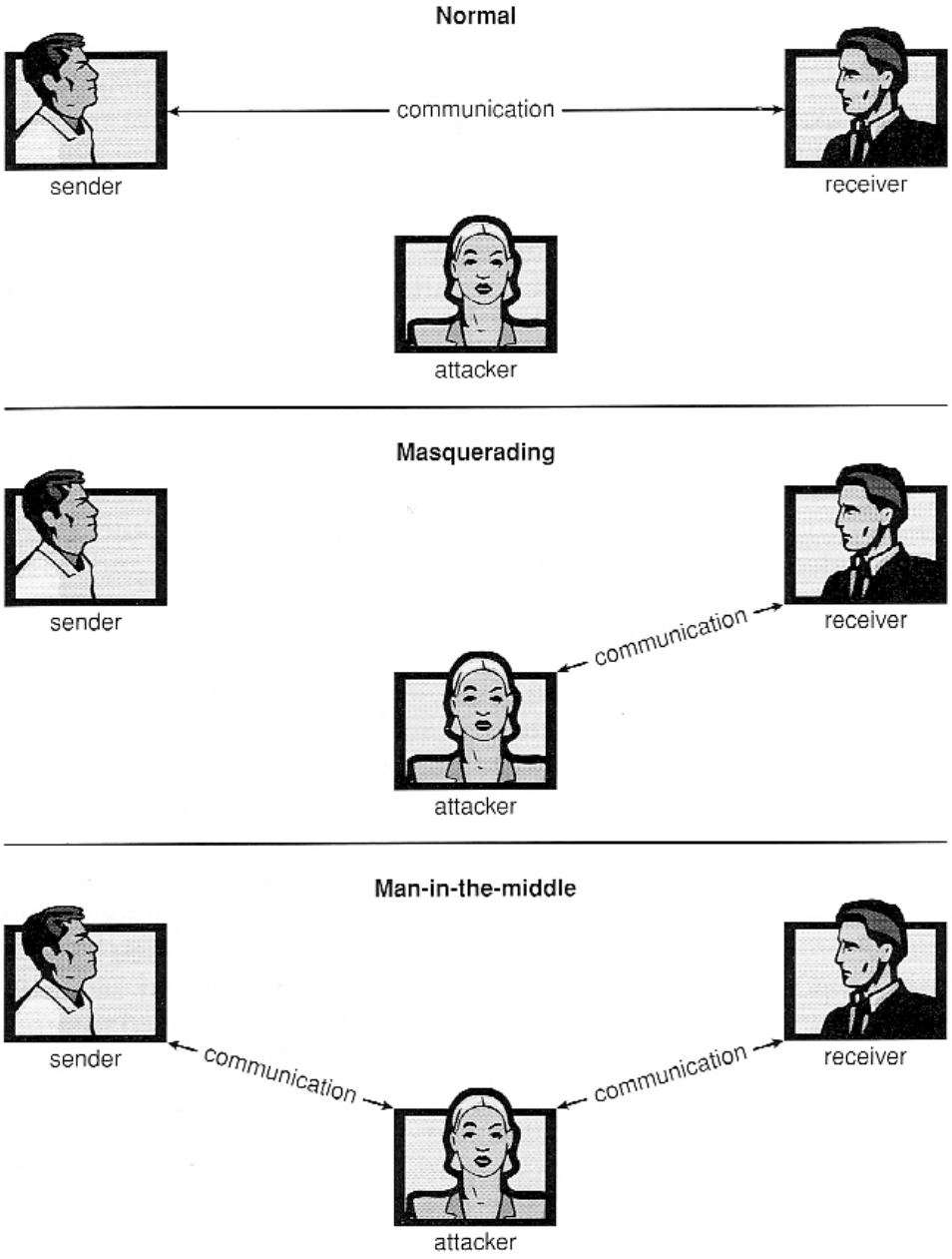


Fig 15.9 Standard Security Attacks

There are four levels at which a system must be protected:

1. **Physical** - The easiest way to steal data is to pocket the backup tapes. Also, access to the root console will often give the user special privileges, such as rebooting the system as root from removable media. Even general access to terminals in a computer room offers some opportunities for an attacker, although today's modern high-speed networking environment provides more and more opportunities for remote attacks.
2. **Human** - There is some concern that the humans who are allowed access to a system be trustworthy, and that they cannot be coerced into breaching security. However more and more attacks today are made via social engineering, which basically

means fooling trustworthy people into accidentally breaching security.

- **Phishing** involves sending an innocent-looking e-mail or web site designed to fool people into revealing confidential information. E.g. spam e-mails pretending to be from e-Bay, PayPal, or any of a number of banks or credit-card companies.

- **Dumpster Diving** involves searching the trash or other locations for passwords that are written down. (Note: Passwords that are too hard to remember, or which must be changed frequently are more likely to be written down somewhere close to the user's station.)

- **Password Cracking** involves divining users passwords, either by watching them type in their passwords, knowing something about them like their pet's names, or simply trying all words in common dictionaries. (Note: "Good" passwords should involve a minimum number of characters, include non-alphabetical characters, and not appear in any dictionary (in any language), and should be changed frequently. Also note that it is proper etiquette to look away from the keyboard while someone else is entering their password.)

3. **Operating System** - The OS must protect itself from security breaches, such as runaway processes (denial of service), memory-access violations, stack overflow violations, the launching of programs with excessive privileges, and many others.

4. **Network** - As network communications become ever more important and pervasive in modern computing environments, it becomes ever more important to protect this area of the system. (Both protecting the network itself from attack, and protecting the local system from attacks coming in through the network.) This is a growing area of concern as wireless communications and portable devices become more and more prevalent.

15.9 SYSTEM AND NETWORK THREATS

Most of the threats described above are termed program threats, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

15.9.1 Worms

A worm is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms

consume system resources, often blocking out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.

One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.

This worm consisted of two parts: A small program called a grappling hook, which was deposited on the target system through one of three vulnerabilities, and the main worm program, which was transferred onto the target system and launched by the grappling hook program.

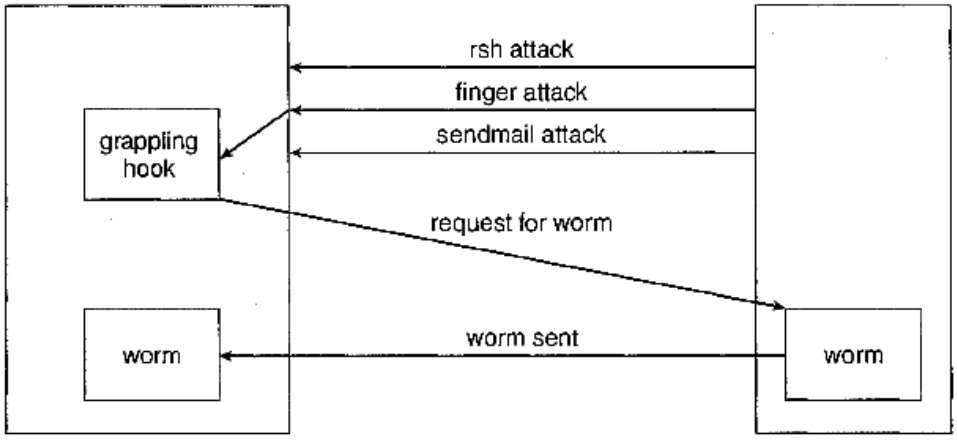


Fig 15.10 The Morris Internet WORM

The three vulnerabilities exploited by the Morris Internet worm were as follows:

1. **rsh (remote shell)** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers (with the same account name on both systems), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were configured so that any user (except root) on system A could access the same account on system B without providing a password.

2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser.

Unfortunately the finger daemon (which ran with system privileges) had the buffer overflow problem, so by sending a special 536-character user name the worm was able to fork a shell on the remote system running with root privileges.

3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.

Once in place, the worm undertook systematic attacks to discover user passwords:

1. First it would check for accounts for which the account name and the password were the same, such as "guest", "guest".
2. Then it would try an internal dictionary of 432 favorite password choices. (I'm sure "password", "pass", and blank passwords were all on the list.)
3. Finally it would try every word in the standard UNIX on-line dictionary to try and break into user accounts.

Once it has access to one or more user accounts, then it would attempt to use those accounts to rsh to other systems, and continue the process.

With each new access the worm would check for already running copies of itself, and 6 out of 7 times if it found one it would stop. (The seventh was to prevent the worm from being stopped by fake copies.)

Fortunately the same rapid network connectivity that allowed the worm to propagate so quickly also quickly led to its demise - Within 24 hours remedies for stopping the worm propagated through the Internet from administrator to administrator, and the worm was quickly shut down.

There is some debate about whether Mr. Morris's actions were a harmless prank or research project that got out of hand or a deliberate and malicious attack on the Internet. However the court system convicted him, and penalized him heavy fines and court costs.

There have since been many other worm attacks, including the W32.Sobig.F@mm attack which infected hundreds of thousands of computers and an estimated 1 in 17 e-mails in August 2003. This worm made detection difficult by varying the subject line

of the infection-carrying mail message, including "Thank You!", "Your details", and "Re: Approved".

15.9.2 Port Scanning

Port Scanning is technically not an attack, but rather a search for vulnerabilities to attack. The basic idea is to systematically attempt to connect to every known (or common or possible) network port on some remote machine, and to attempt to make contact. Once it is determined that a particular computer is listening to a particular port, then the next step is to determine what daemon is listening, and whether or not it is a version containing a known security flaw that can be exploited.

Because port scanning is easily detected and traced, it is usually launched from zombie systems, i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner. For this reason it is important to protect "innocuous" systems and accounts as well as those that contain sensitive information or special privileges.

There are also port scanners available that administrators can use to check their own systems, which report any weaknesses found but which do not exploit the weaknesses or cause any problems. Two such systems are nmap (http://www.insecure.org/nmap) and nessus (http://www.nessus.org). The former identifies what OS is found, what firewalls are in place, and what services are listening to what ports. The latter also contains a database of known security holes, and identifies any that it finds.

15.9.3 Denial of Service

Denial of Service (DOS) attacks do not attempt to actually access or damage systems, but merely to clog them up so badly that they cannot be used for any useful work. Tight loops that repeatedly request system services are an obvious form of this attack.

DOS attacks can also involve social engineering, such as the Internet chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", which clogs up not only the Internet mail system but also the web server to which everyone is directed. (Note: Sending a "reply all" to such a message notifying everyone that it was just a hoax also clogs up the Internet mail service, just as effectively as if you had forwarded the thing.)

Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts.

15.10 IMPLEMENTING SECURITY DEFENSES

15.10.1 SECURITY POLICY

A security policy should be well thought-out, agreed upon, and contained in a living document that everyone adheres to and is updated as needed. Examples of contents include how often port scans are run, password requirements, virus detectors, etc.

15.10.2 VULNERABILITY ASSESSMENT

Periodically examine the system to detect vulnerabilities.

- Port scanning.
- Check for bad passwords.
- Look for suid programs.
- Unauthorized programs in system directories.
- Incorrect permission bits set.
- Program checksums / digital signatures which have changed.
- Unexpected or hidden network daemons.
- New entries in startup scripts, shutdown scripts, cron tables, or other system scripts or configuration files.
- New unauthorized accounts.

The government considers a system to be only as secure as its most far-reaching component. Any system connected to the Internet is inherently less secure than one that is in a sealed room with no external communications.

Some administrators advocate "security through obscurity", aiming to keep as much information about their systems hidden as possible, and not announcing any security concerns they come across. Others announce security concerns from the rooftops, under the theory that the hackers are going to find out anyway, and the only one kept in the dark by obscurity are honest administrators who need to get the word.

15.10.3 INTRUSION DETECTION

Intrusion detection attempts to detect attacks, both successful and unsuccessful attempts. Different techniques vary along several axes:

- The time that detection occurs, either during the attack or after the fact.
- The types of information examined to detect the attack(s). Some attacks can only be detected by analyzing multiple sources of information.

- The response to the attack, which may range from alerting an administrator to automatically stopping the attack (e.g. killing an offending process), to tracing back the attack in order to identify the attacker.

Intrusion Detection Systems, IDSs, raise the alarm when they detect an intrusion. Intrusion Detection and Prevention Systems, IDPs, act as filtering routers, shutting down suspicious traffic when it is detected.

There are two major approaches to detecting problems:

1. **Signature-Based Detection** scans network packets, system files, etc. looking for recognizable characteristics of known attacks, such as text strings for messages or the binary code for "exec /bin/sh". The problem with this is that it can only detect previously encountered problems for which the signature is known, requiring the frequent update of signature lists.

2. **Anomaly Detection** looks for "unusual" patterns of traffic or operation, such as unusually heavy load or an unusual number of logins late at night.

The benefit of this approach is it can detect previously unknown attacks, so called zero-day attacks. To be effective, anomaly detectors must have a very low false alarm (false positive) rate, lest the warnings get ignored, as well as a low false negative rate in which attacks are missed.

15.10.4 VIRUS PROTECTION

Modern anti-virus programs are basically signature-based detection systems, which also have the ability (in some cases) of disinfecting the affected files and returning them back to their original condition.

Both viruses and anti-virus programs are rapidly evolving. For example viruses now commonly mutate every time they propagate, and so anti-virus programs look for families of related signatures rather than specific ones. Some antivirus programs look for anomalies, such as an executable program being opened for writing (other than by a compiler.)

Avoiding bootleg, free, and shared software can help reduce the chance of catching a virus, but even shrink-wrapped official software has on occasion been infected by disgruntled factory workers. Some virus detectors will run suspicious programs in a sandbox, an isolated and secure area of the system which mimics the real system.

Rich Text Format, RTF, files cannot carry macros, and hence cannot carry Word macro viruses. Known safe programs (e.g. right after a fresh install or after a thorough examination) can be digitally

signed, and periodically the files can be re-verified against the stored digital signatures. (Which should be kept secure, such as on off-line write-only medium)

15.10.5 AUDITING, ACCOUNTING, AND LOGGING

Auditing, accounting, and logging records can also be used to detect anomalous behavior.

Some of the kinds of things that can be logged include authentication failures and successes, logins, running of suid or sgid programs, network accesses, system calls, etc. In extreme cases almost every keystroke and electron that moves can be logged for future analysis. (Note that on the flip side, all this detailed logging can also be used to analyze system performance. The down side is that the logging also affects system performance (negatively), and so a Heisenberg effect applies).

"The Cuckoo's Egg" tells the story of how Cliff Stoll detected one of the early UNIX break-ins when he noticed anomalies in the accounting records on a computer system being used by physics researchers.

15.11 LET US SUM UP

- A computer can be viewed as a collection of processes and objects (both hardware and software).
- A protection domain specifies the resources that a process may access.
- A domain is defined as a set of < object, { access right set } > pairs.
- Domains may be realized in different fashions - as users, or as processes, or as procedures.
- Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running.
- The ability to copy rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object.
- Role-Based Access Control, RBAC, assigns privileges to users, programs, or roles as appropriate, where "privileges" refer to the right to call certain system calls, or to use certain parameters with those calls.
- Hydra is a capability-based system that includes both system-defined rights and user-defined rights.

- In a compiler-based approach to protection enforcement, programmers directly specify the protection needed for different resources at the time the resources are declared.
- Even if the underlying OS does not provide advanced protection mechanisms, the compiler can still offer some protection, such as treating memory accesses differently in code versus data segments.
- The security of the compiler-based enforcement is dependent upon the integrity of the compiler itself, as well as requiring that files not be modified after they are compiled.
- Java was designed from the very beginning to operate in a distributed environment, where code would be executed from a variety of trusted and untrusted sources.
- When a Java program runs, it load up classes dynamically, in response to requests to instantiates objects of particular types.
- A worm is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system
- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988
- Port Scanning is technically not an attack, but rather a search for vulnerabilities to attack
- Because port scanning is easily detected and traced, it is usually launched from zombie systems, i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner.
- Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts
- Any system connected to the Internet is inherently less secure than one that is in a sealed room with no external communications.
- Avoiding bootleg, free, and shared software can help reduce the chance of catching a virus, but even shrink-wrapped official software has on occasion been infected by disgruntled factory workers.

15.12 UNIT END QUESTIONS

12. Write short notes on : (a) Access Matrix (b) Hydra
13. What are the most common types of violations?
14. Describe in brief about protection in JAVA.
15. State and explain the levels at which a system must be protected.

16. What is Port Scanning.
17. Define :
    a. Grappling
    b. Zero-day attacks
18. How are security defenses implemented?