Unit 7. Operative Systems

Contenido

1 Introduction

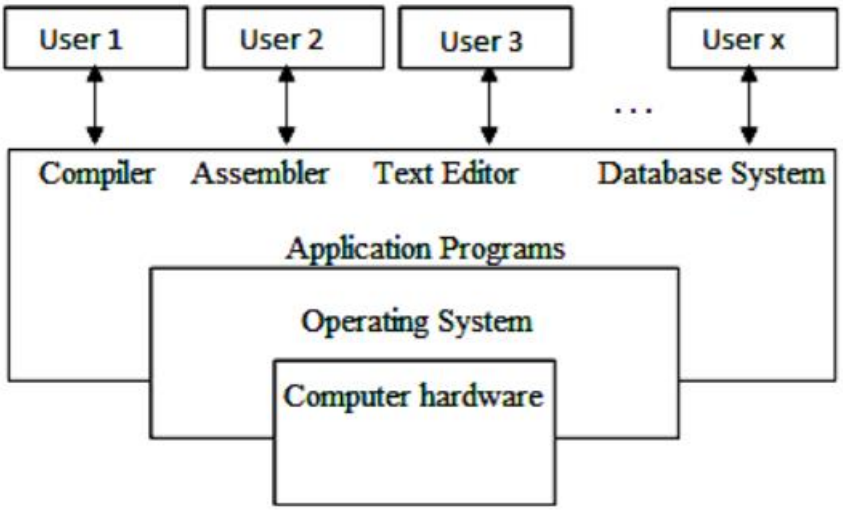
Each user has his own personal thoughts on what the computer system is for. The operating system, or OS, as we will often call it, is the intermediary between users and the computer system. It provides the services and features present in abstract views of all its users through the computer system. An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

1.1 *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)

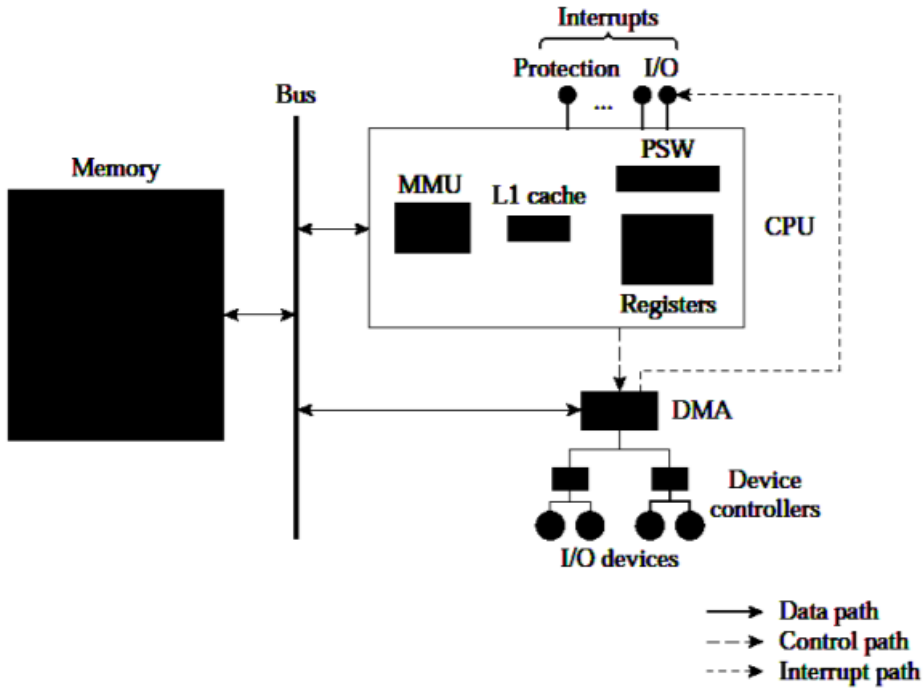
- disks (a particular kind of secondary storage)
- network interface
- display
- keyboard
- mouse An operating system can also be commonly defined as “a program running at all times on the computer (usually called the kernel), with all other being application programs”.



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

1.2 *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.



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.

1.3 *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. Next Table 1.1 lists 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, roundrobin 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

1.3.1 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 mid1950s 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.

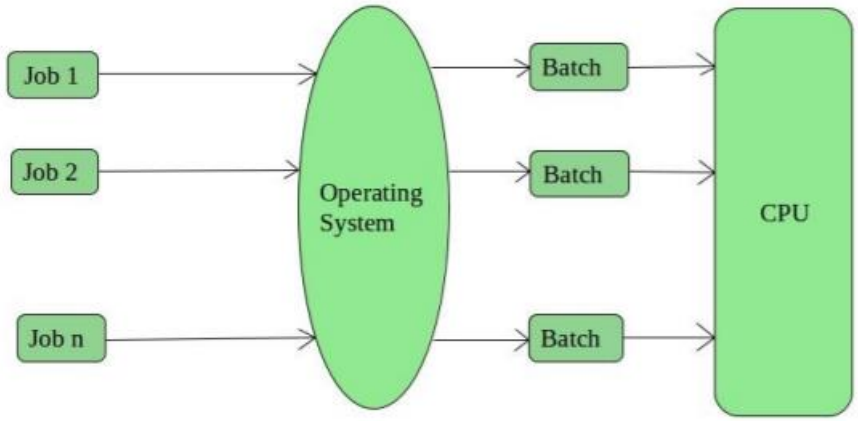
In a batch processing operating system, the prime concern is CPU efficiency. The batch processing system operates in a strict one job-at-a-time manner; within a job, it executes the programs one after another. Thus only one program is under execution at any time.

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.

1.3.1.1 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.

This type of operating system does not interact with the computer directly. There is an operator which takes similar jobs having same requirement and group them into batches. It is the responsibility of operator to sort the jobs with similar needs.



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

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

System utilization example

1.3.1.2 MULTI-PROGRAMMED BATCH SYSTEMS

Multiprogramming operating systems are fairly sophisticated compared to single-program, or uni-programming, 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.

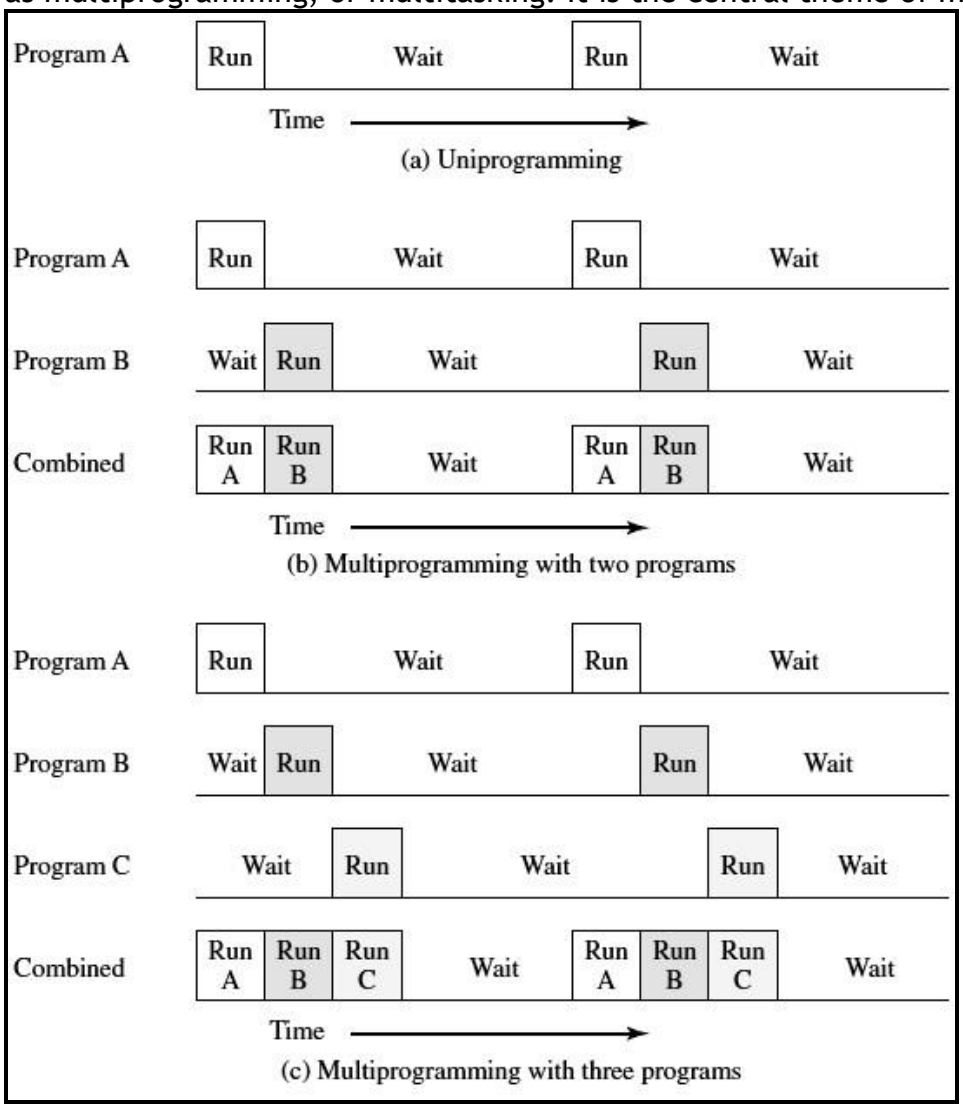


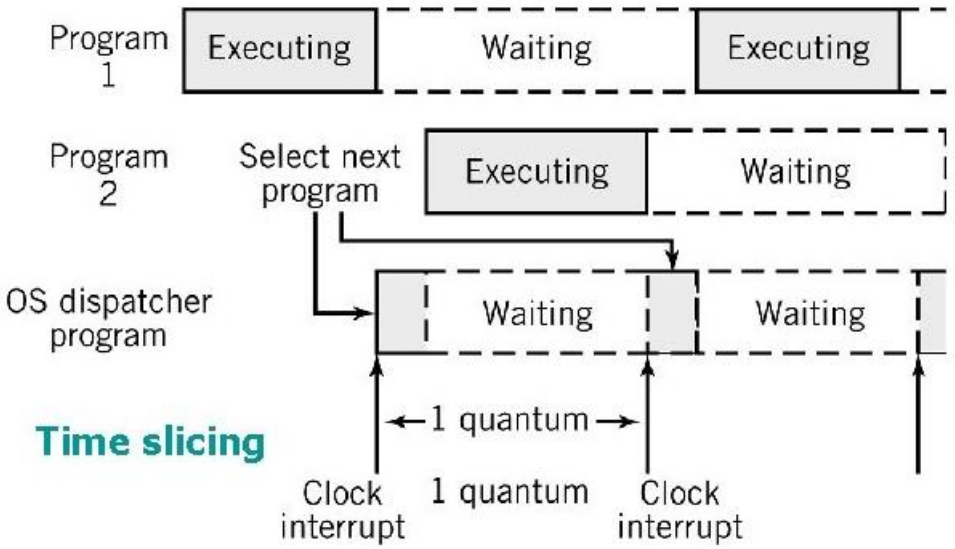
Fig Multiprogramming Example

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.

1.3.2 TIME SHARING SYSTEMS

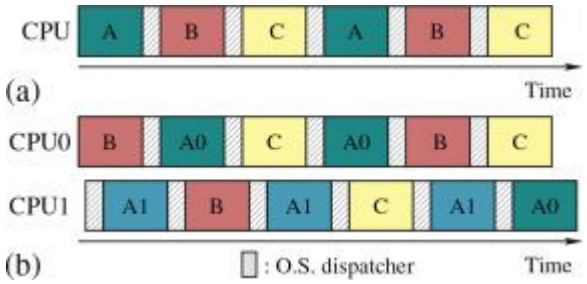
A time-sharing operating system focuses on facilitating quick response to *subrequests* made by all processes, which provides a tangible benefit to users. It is achieved by giving a fair execution opportunity to each process through two means: The OS services all processes by turn, which is called round-robin 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.



1.3.3 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.



1.3.3.1 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.

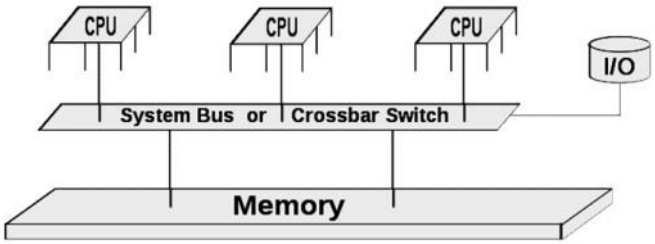


Fig A typical SMP system

1.3.3.2 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.

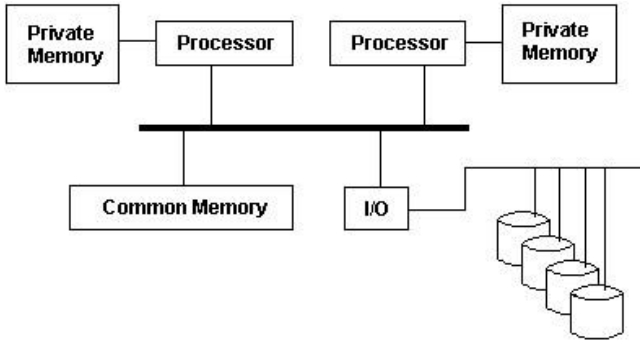


Fig Multiple processors with unique access to memory and I/O

1.3.4 REAL TIME SYSTEMS

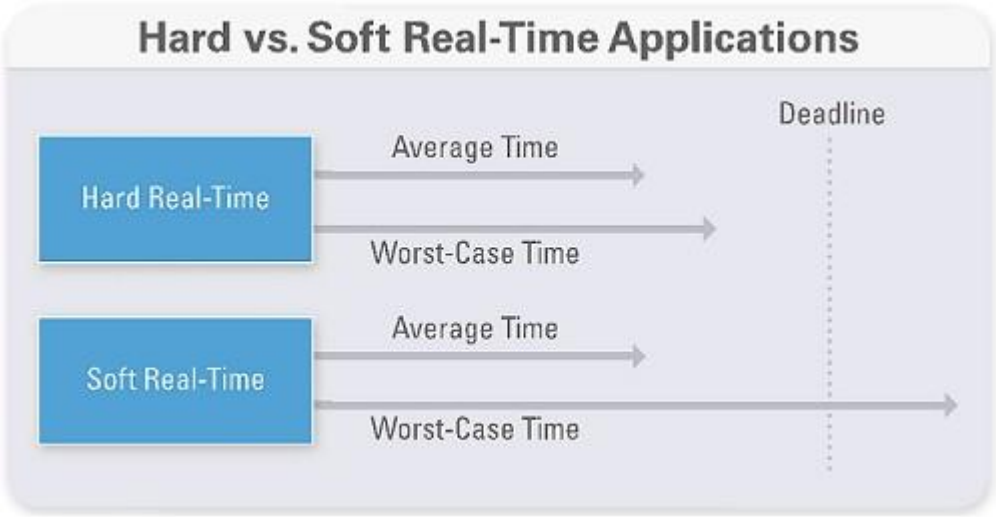
A real-time operating system is used to implement a computer application for controlling or tracking of real-world activities. The application needs to complete its computational tasks in a timely manner to keep abreast of external events in the activity that it controls. To facilitate this, the OS permits 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. An example of a hard real time system should be an autopilot for a plane or a car. An example of a soft real-time system could be a control system for temperature in a room.

1.3.4.1 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 probably 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.



This kind of systems usually react to external events, as for example a red light should detain your car. Some of these events require a prompt response the system. In other words, they need to be handled in time, and they are tagged as more prioritary in the system. For instance, imagine your car do not follow the traffic signs. Providing that it is highly likely that a traffic violation causes an accident, the autopilot system must respond in time, and meet the deadline. On the contrary, to control the temperature of a room, there is no need for such a quick reaction, and the system can switch on the heater after the deadline.

1.3.4.2 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.

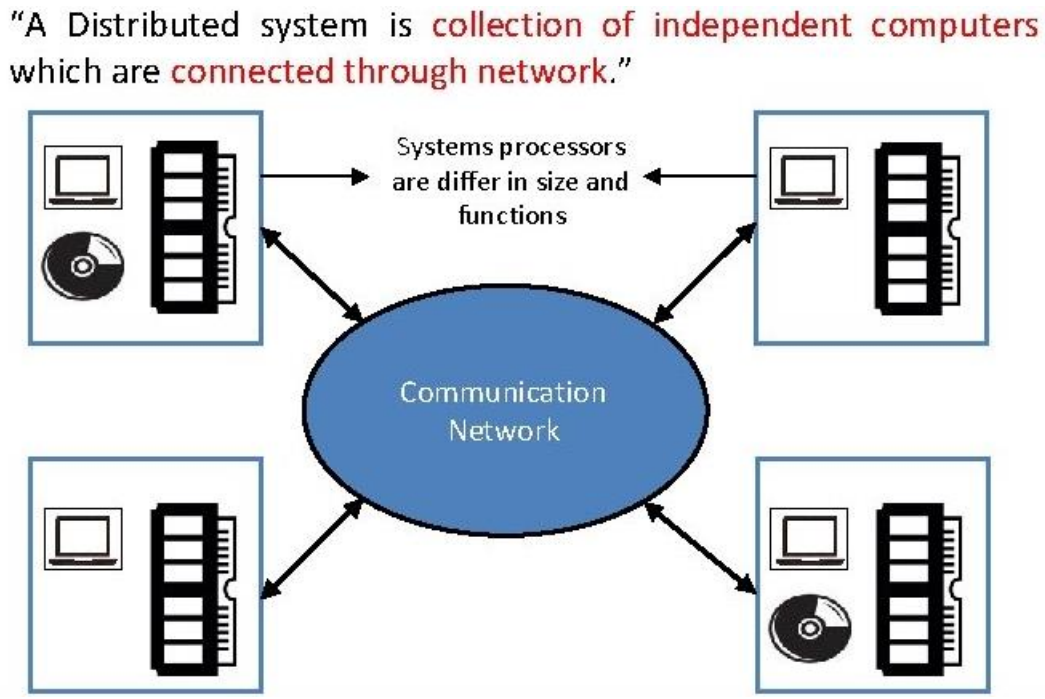
Reliability	The OS ensures that an application can continue to function even when faults occur in the computer.

1.3.5 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 named 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.

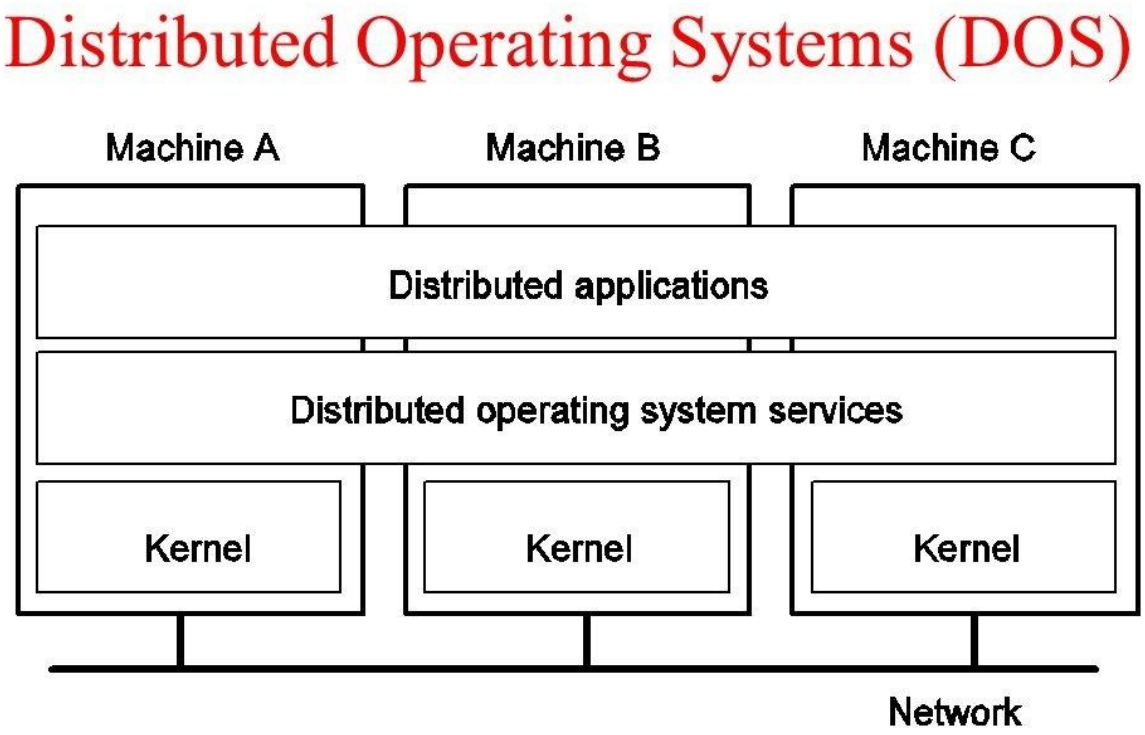
A distributed operating system is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. The users may not be aware of where their programs are being run or where their files are located; that should all be handled automatically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



A distributed operative system shared their components and services among different machines. Nevertheless, the distribution of responsibilities (applications and services) does not need to be homogeneous because different machines can hold different workload. Besides, the idea of a Distributed Operative System is to functionally works giving the

impression to the user that he is managing a traditional operative system.



1.3.6 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.

1.3.7 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 multi-gigabyte hard disks, which changes a lot.
Two of the most popular operating systems for handhelds are Symbian OS and Android OS.

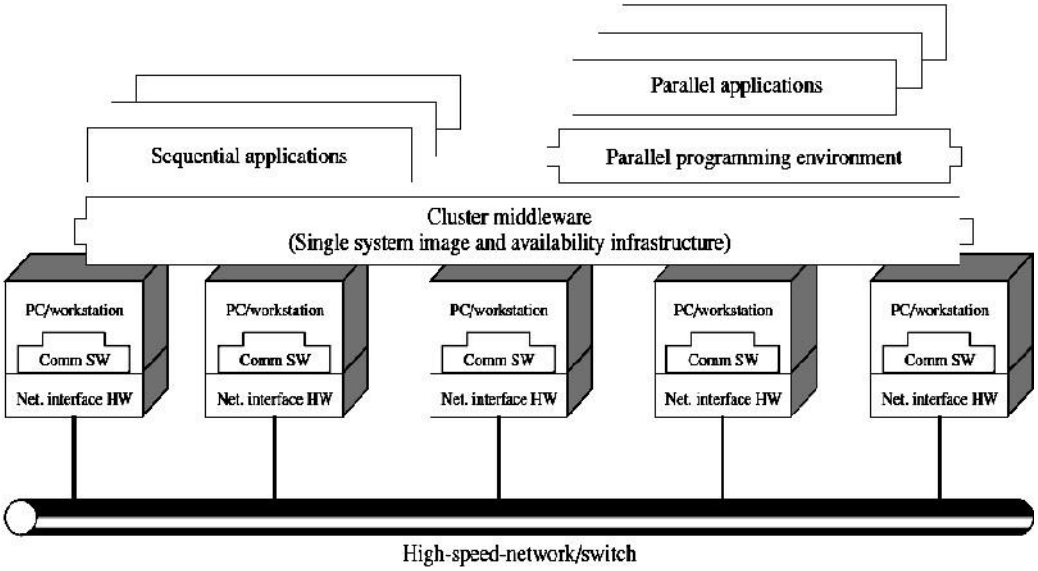
1.3.8 CLUSTERED SYSTEMS

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.

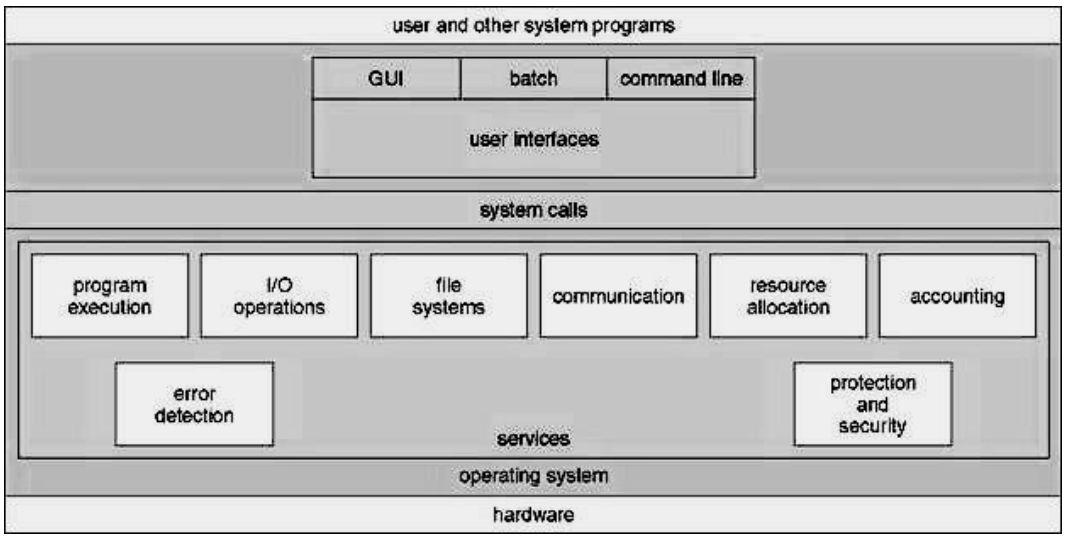
In asymmetric clustering, one machine is in hot standby mode while the other is running the applications. The hot standby host (machine) does nothing but monitor the active server. If that server fails, the hot standby host becomes the active server. In symmetric mode, two or more hosts are running applications, 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**

2 OS SERVICES, CALLS, INTERFACES AND PROGRAMS

An operating system is made up of many modules. Besides, an operative systems offer services to the applications and the user interfaces. This services allow to fully use the computer system. Since the file system to the mouse or the internet (communications) are controlled by the Operative System. Providing that any application or the user want to accede to a file, use the mouse or connect to any give websites, they must do it through calls to the operative system. This calls make use of the OS services depending on which action is performed.



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:

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).

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.

FILE-SYSTEMS:

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

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..

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.

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.

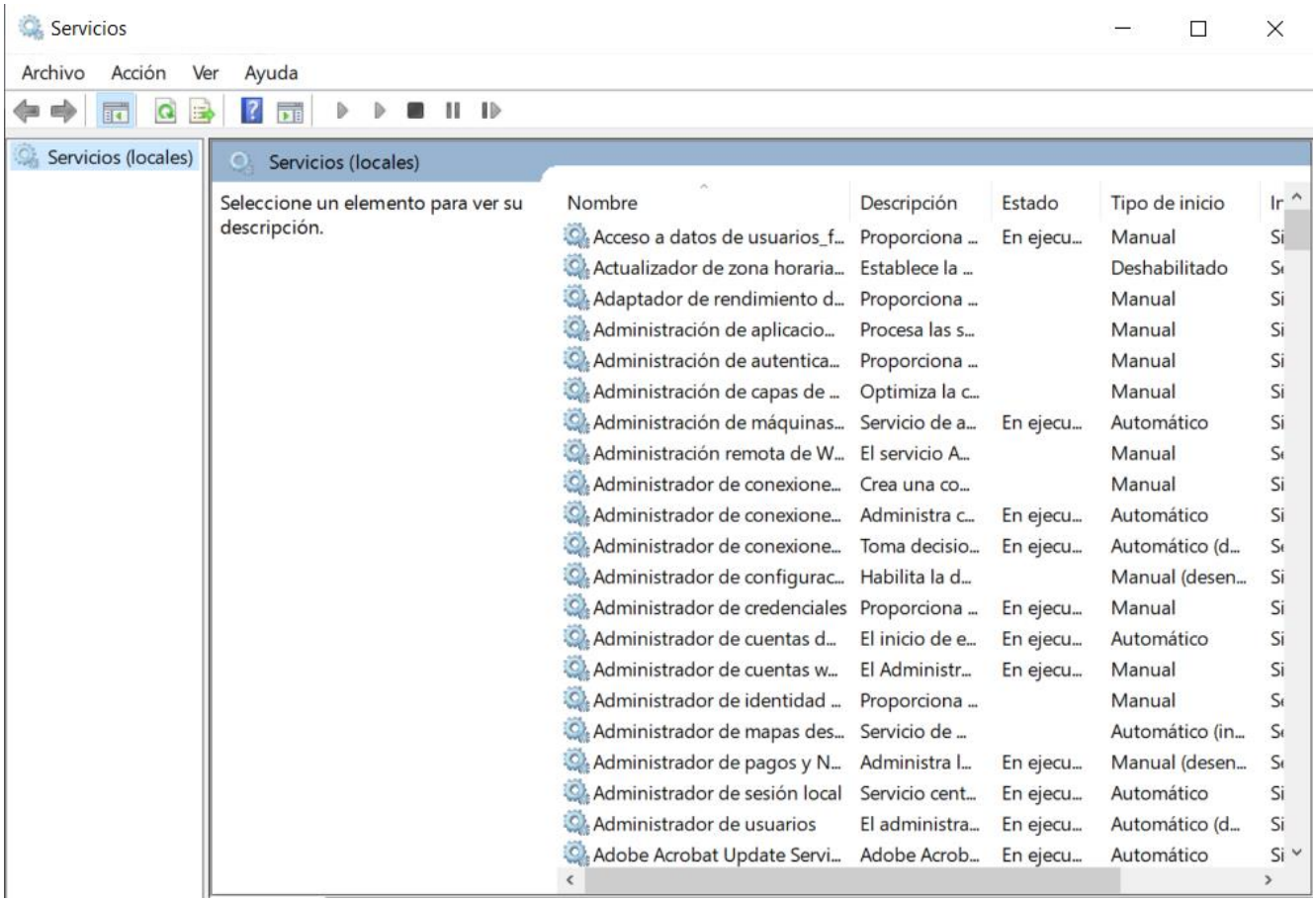
ERROR DETECTION:

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.

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.



2.1 *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.

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 MSDOS 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.

```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19042.1288]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\carlo>dir
 El volumen de la unidad C es Windows
 El número de serie del volumen es: 5071-71CF

 Directorio de C:\Users\carlo

21/10/2021  16:47    <DIR>          .
21/10/2021  16:47    <DIR>          ..
08/01/2021  11:54    <DIR>          .afirma
03/11/2021  08:37    <DIR>          .android
13/11/2020  15:53    <DIR>          .AndroidStudio4.0
03/05/2021  10:31    <DIR>          .AndroidStudio4.1
15/11/2020  19:31               157 .appletviewer
18/10/2021  19:35             7.907 .bash_history
20/02/2021  19:50            21.222 .boto
03/02/2021  22:31    <DIR>          .cache
18/10/2021  21:24    <DIR>          .config
18/10/2021  21:59    <DIR>          .docker
23/09/2021  11:00    <DIR>          .dotnet
16/02/2021  16:32    <DIR>          .eclipse
13/11/2020  16:25                16 .emulator_console_auth_token
14/12/2020  17:49                53 .git-for-windows-updater
18/10/2021  21:22               608 .gitconfig
24/10/2021  17:41    <DIR>          .gradle
07/12/2020  18:58    <DIR>          .lemminx
15/02/2021  01:19    <DIR>          .m2
16/02/2021  00:11    <DIR>          .nbi
```

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).

2.2 *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).

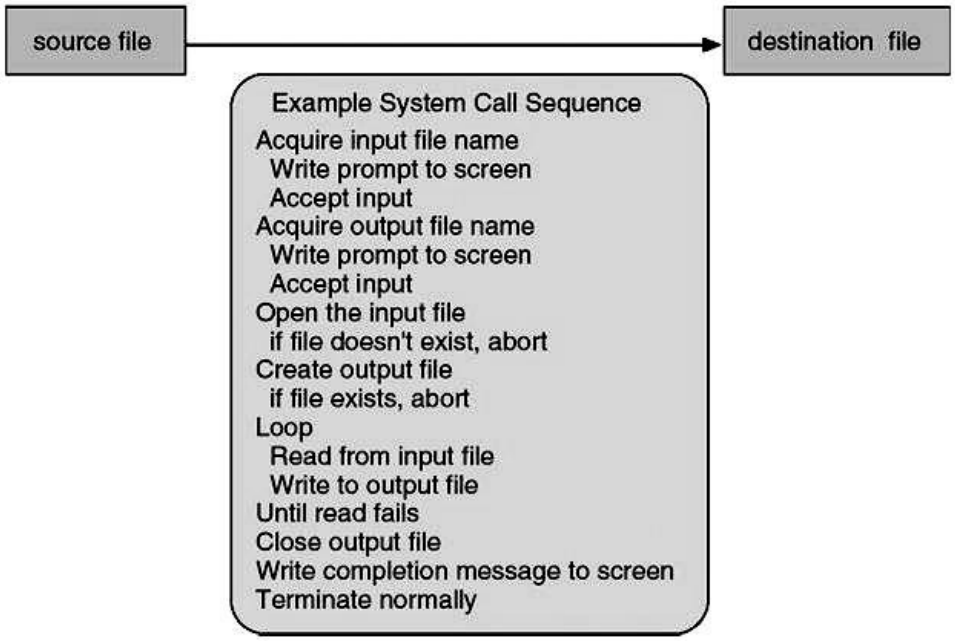


Fig 3.3 Example of System Calls

Call number	Call name	Description
1	exit	Terminate execution of This program
3	read	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	system	sethostname Set hostname of the computer
78	gettimeofday	Get time of day
79	settimeofday	Set time of day

Table Some Linux System Calls

2.2.1 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.

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

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

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

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

COMMUNICATIONS
Create, delete communication connection
Send, receive messages

Transfer status information
Attach or detach remote devices

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

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

Table Examples of Windows and UNIX System Calls

2.3 *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:

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

.

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.

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.

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.

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.

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.

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.

2.4 *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.

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.

3 OPERATING SYSTEM STRUCTURES

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.

3.1 *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.

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.

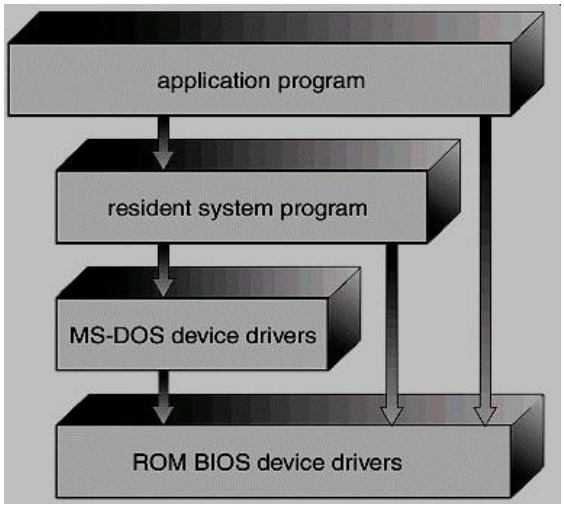
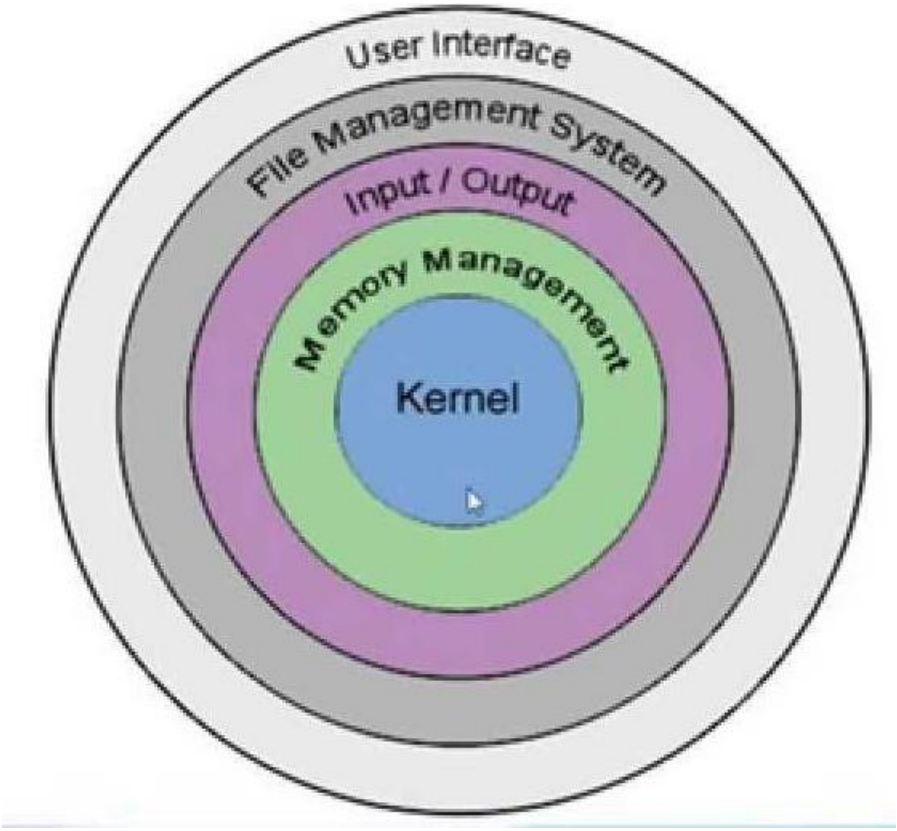


Fig MS-DOS LAYER STRUCTURE

LAYERED APPROACH:

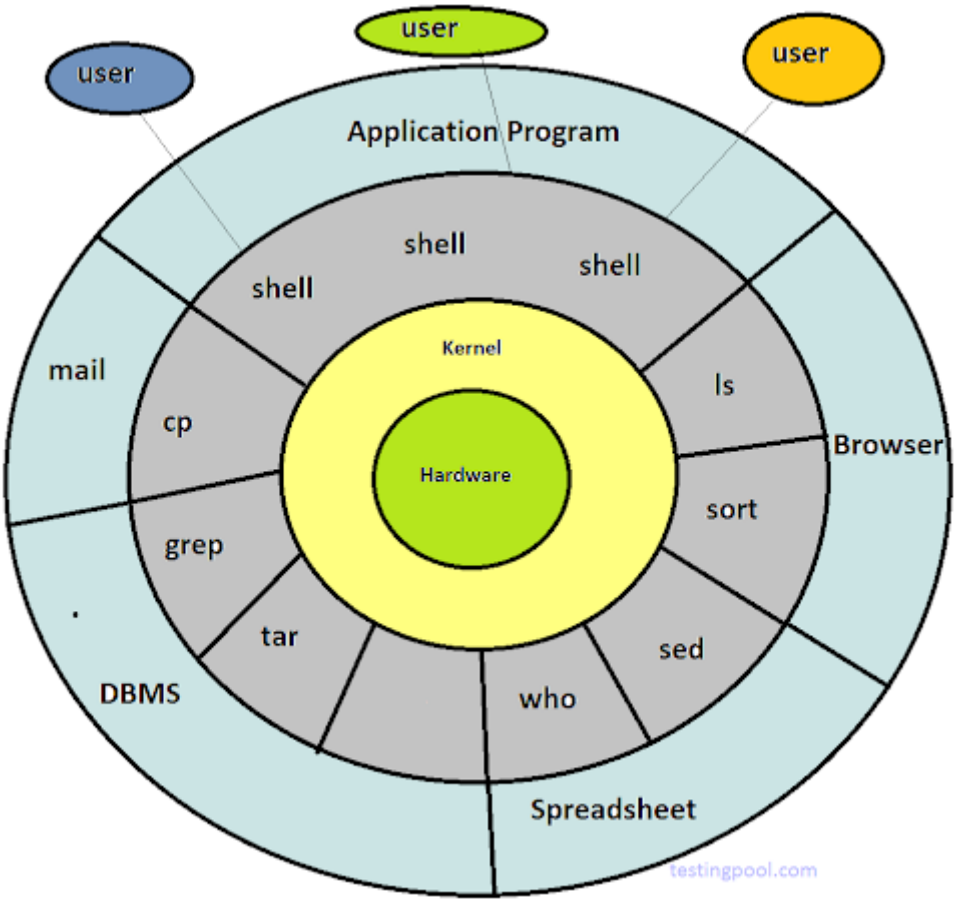
In layered approach, the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware, the highest (layer N) is the user interface. An operating system layer is an implementation of an abstract object made up of data and the operations that can manipulate those 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.


user
user
user
Application Program
shell
shell
shell
Kernel
Hardware
mail
cp
ls
Browser
sort
grep
tar
sed
who
DBMS
Spreadsheet
testingpool.com

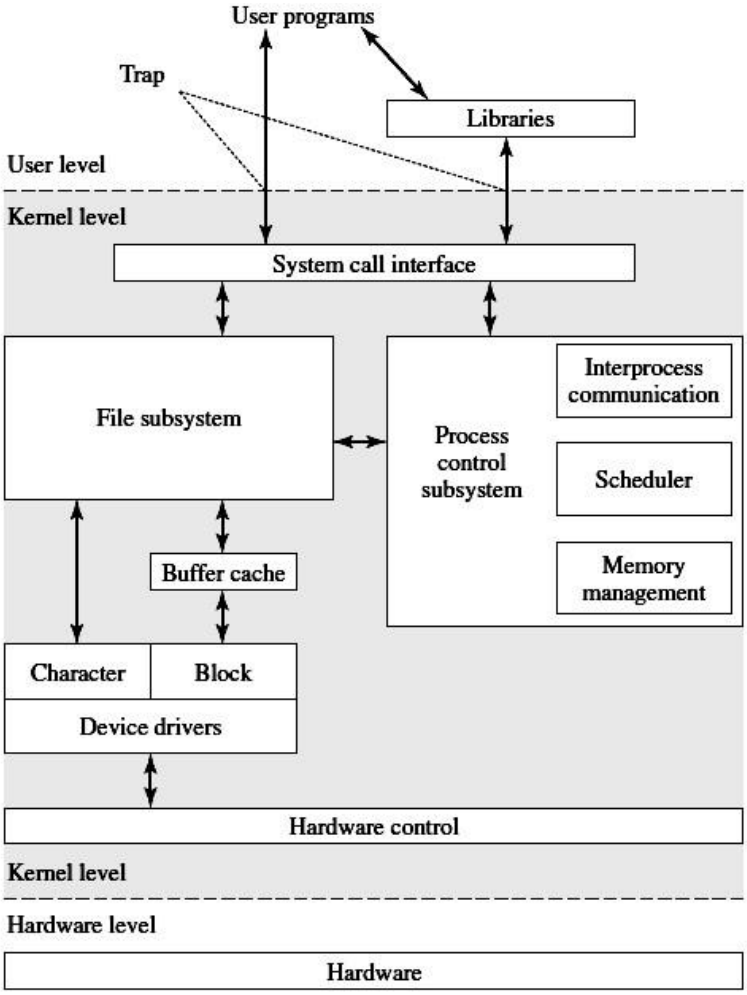


Fig Traditional UNIX Kernel

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.

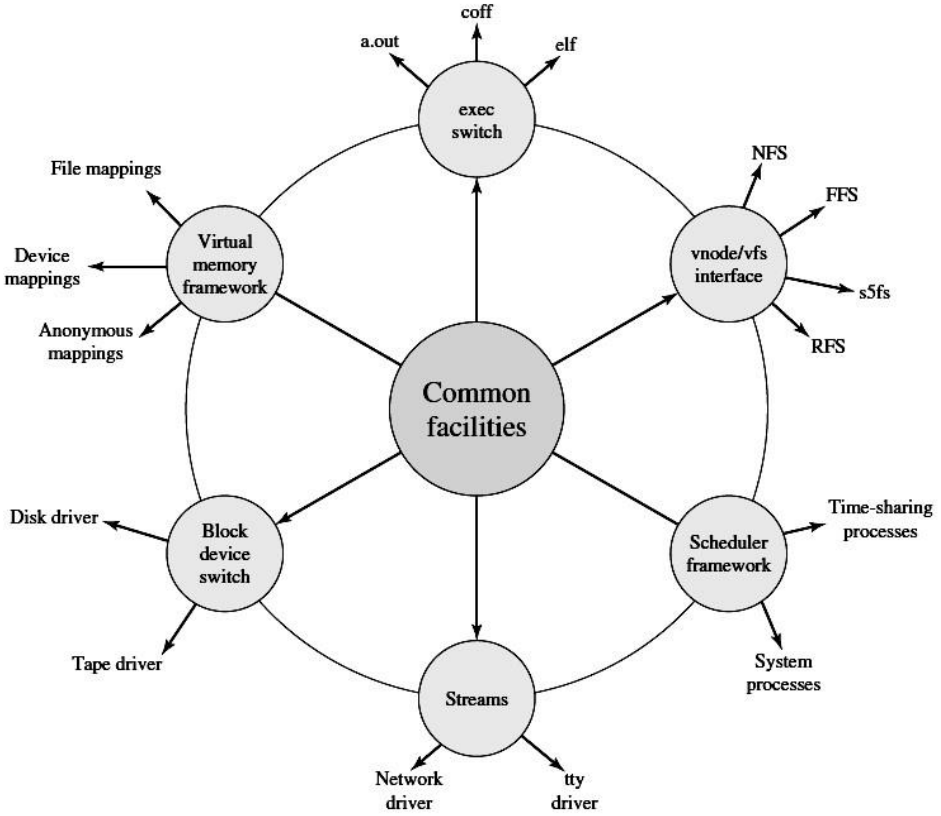


Fig Modern UNIX Kernel

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 UNIXlike 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.

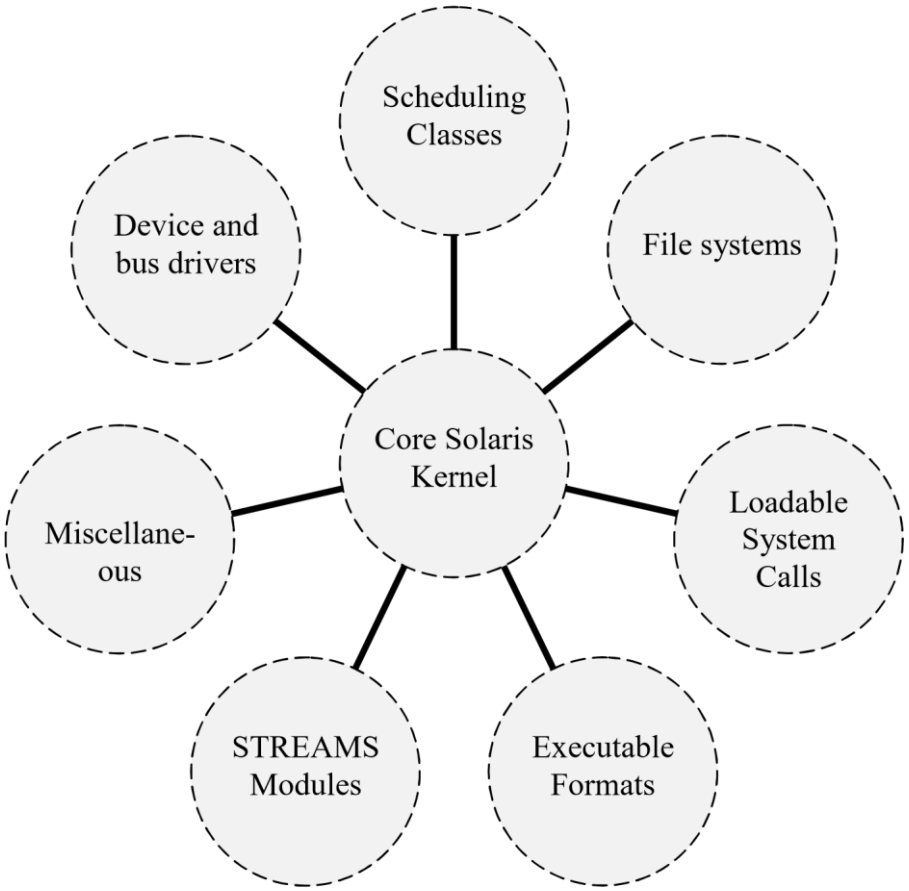


Fig Solaris Loadable Modules

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.

3.2 *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.

3.3 *SYSTEM BOOT*

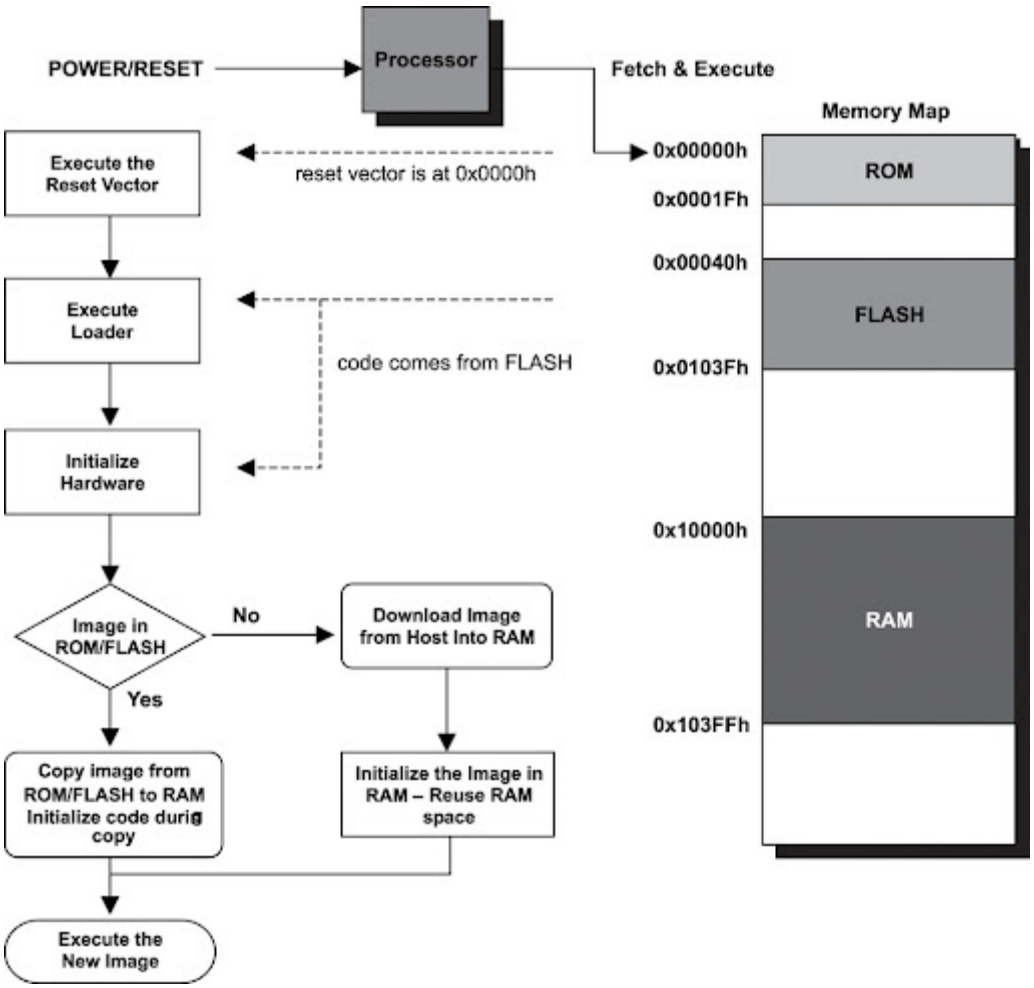
The general approach when most computers boot up goes something like this:

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.

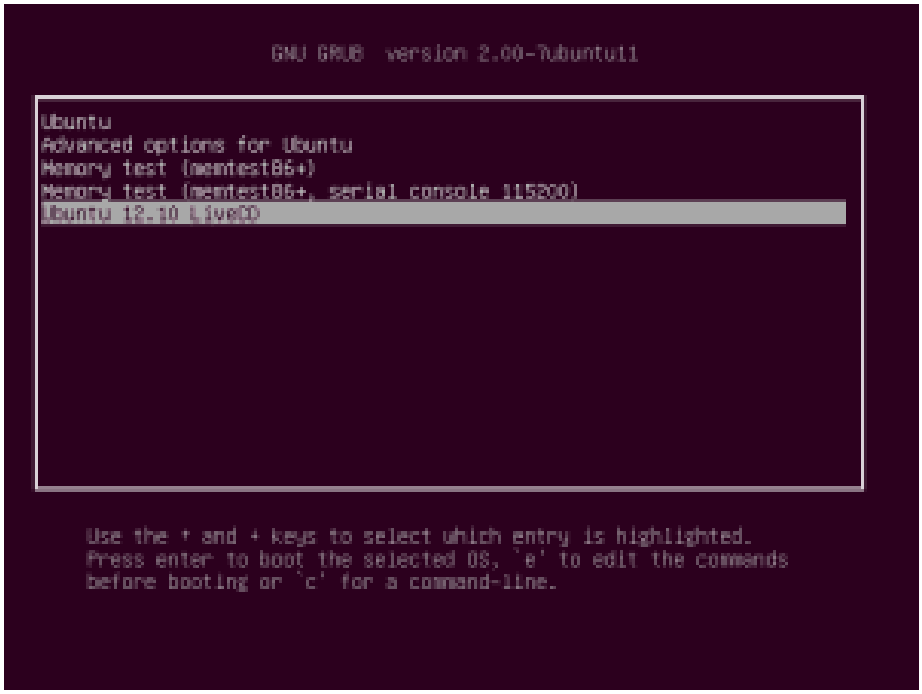
Assuming the utility has not been invoked, the bootstrap program then looks for a non-volatile storage device containing an OS. Depending on configuration, it may look for a pen drive, CD /DVD ROM drive, or primary or secondary hard drives, in the order specified by the HW configuration utility.

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 does not 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.)

Questions

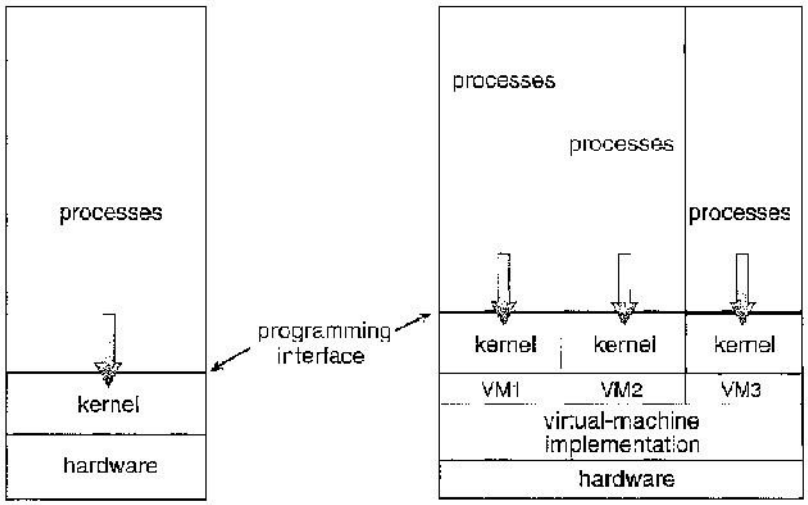
1) What are the differences between layered approach and microkernel approach?
2) What information is needed to configure an OS?
3) What do you mean by System Boot?
4) Define:
    a. Kernel loadable modules
    b. Maintenance mode

4 VIRTUAL MACHINES

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.

4.1 *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.



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

History

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

4.1.1 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.

4.1.2 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 paravirtual 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.

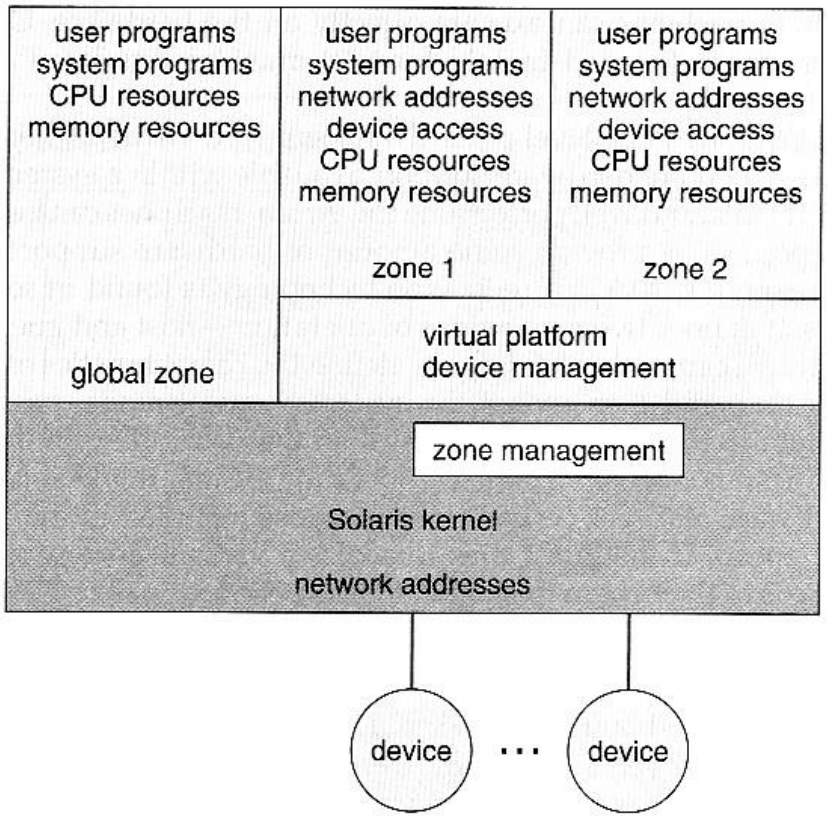


Fig Solaris 10 with two containers

4.1.3 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).

4.1.4 Examples

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.

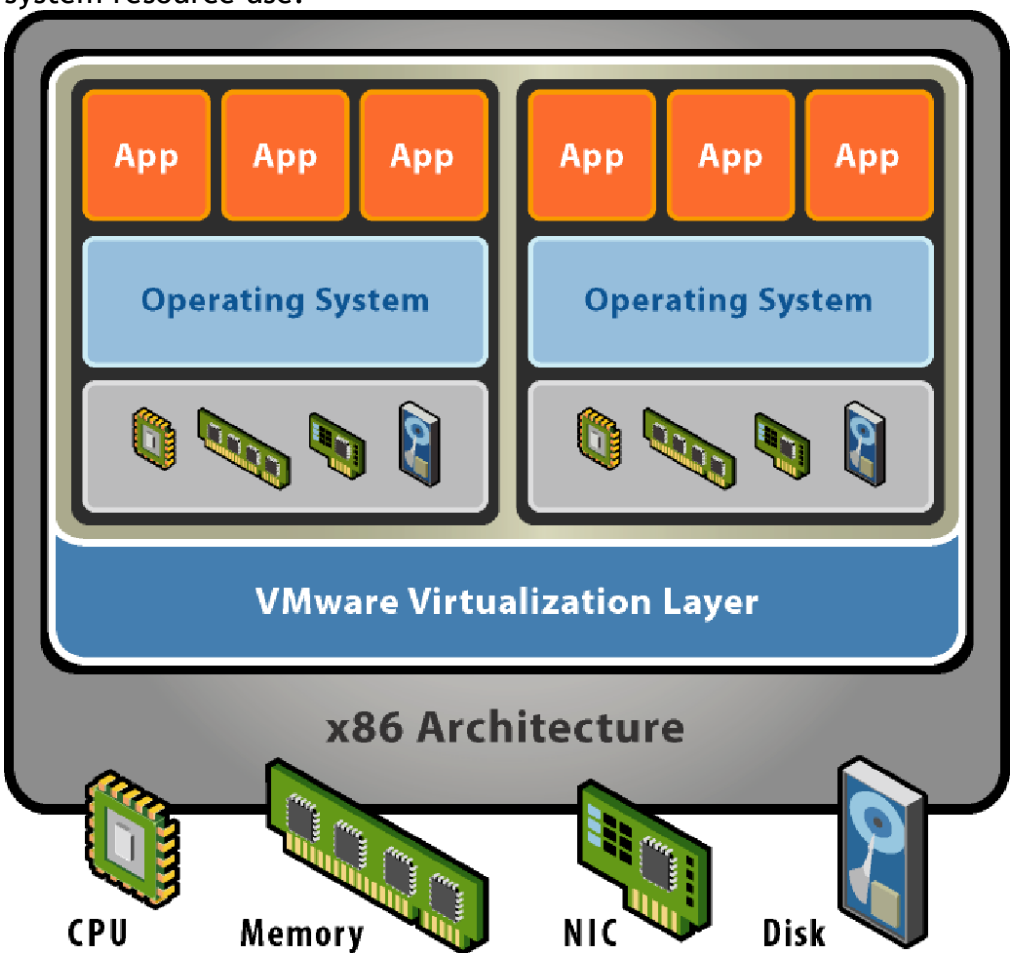


Fig VMware Architecture

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.

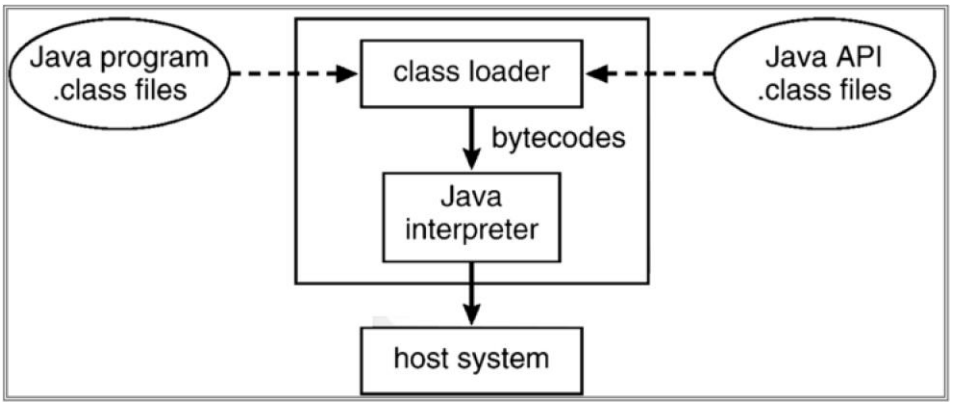


Fig The JAVA Virtual Machine THE .NET FRAMEWORK

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 Architecture of the CLR.

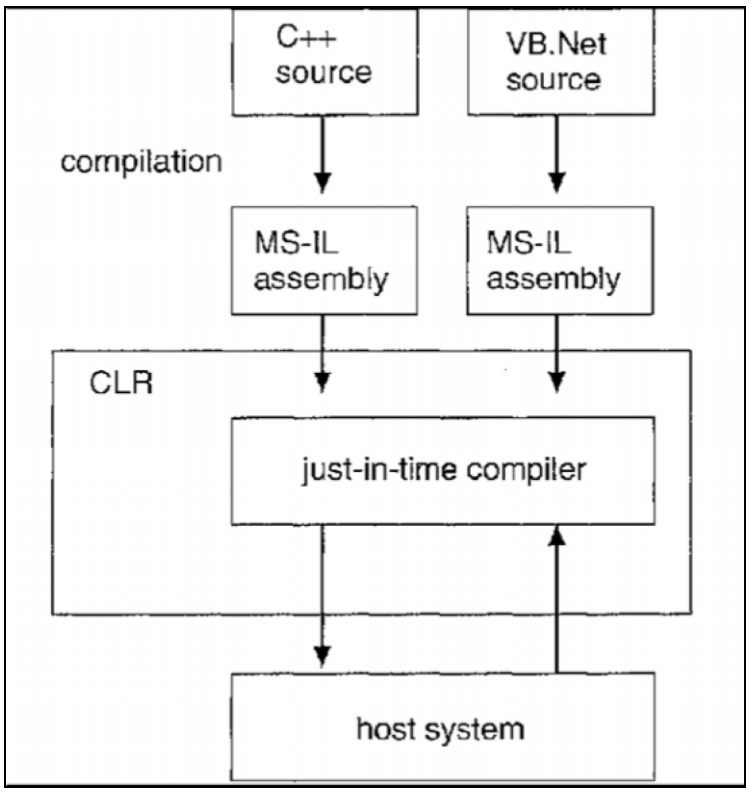


Fig Architecture of the CLR for the .NET Framework

UNIT END QUESTIONS

1) Write a short note on Virtual Machines.
2) Define : (a) Para-virtualization (b) JVM
3) Describe the architecture of :
   (a) The JAVA Virtual Machine
   (b) CLR for the .NET Framework

5 PROCESS

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 inter-process communication and user creation of processes.

5.1 *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.

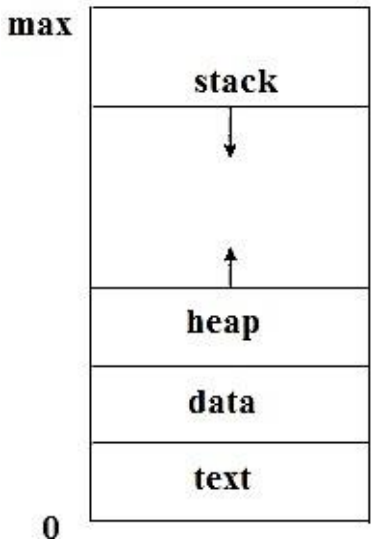


Fig Process in Memory

5.1.1 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

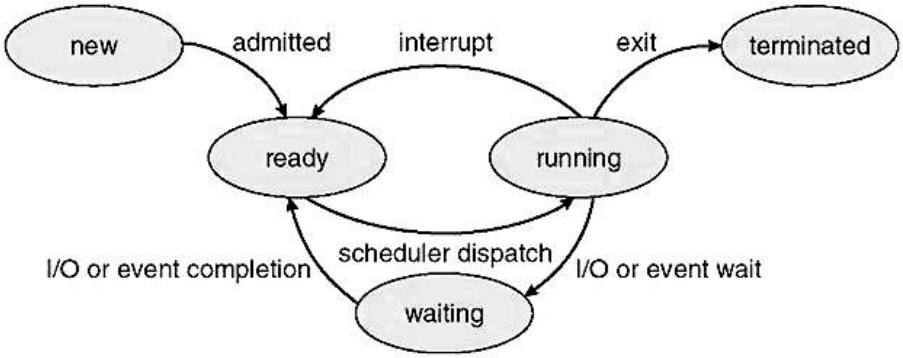


Fig Block diagram of Process States

5.1.2 PROCESS CONTROL BLOCK

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

Fig 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.

5.1.3 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.

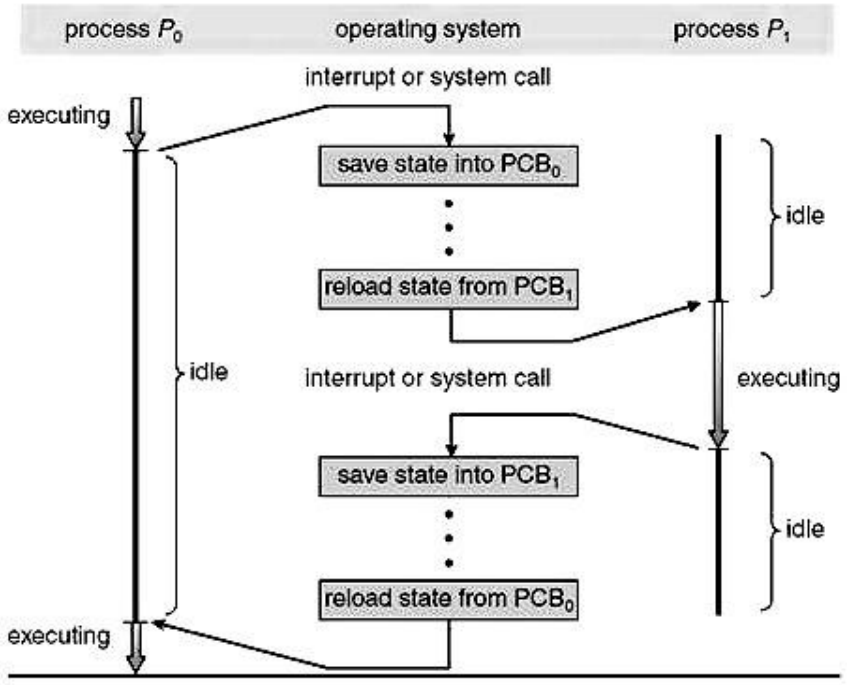
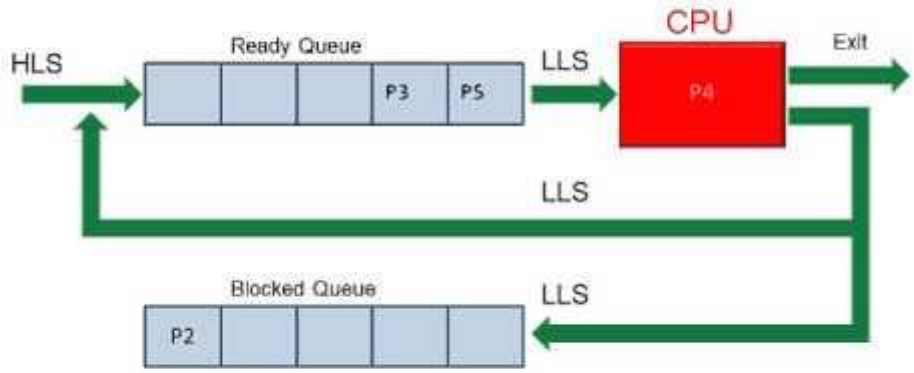


Fig CPU switch from process to process

5.2 *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***.



5.3 *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

Questions

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

6 PROCESS SCHEDULING ALGORITHMS

Scheduling algorithms in Modern Operating Systems are used to:

- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time

6.1 *SCHEDULING ALGORITHMS*

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

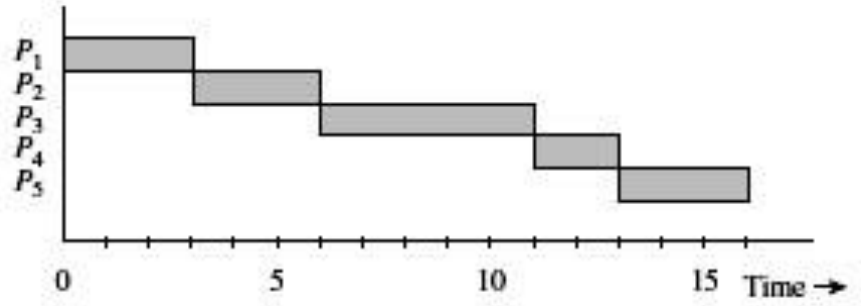
Requests are scheduled in the order in which they arrive in the system. The list of pending requests is organized as a queue. The scheduler always schedules the first request in the list. An example of FCFS scheduling is a batch processing system in which jobs are ordered according to their arrival times (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 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 6.1 Scheduling using FCFS policy**

Figure 6.1 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.

6.1.2 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.

Job	Duration (minutes)
A	8
B	4

C	4
D	4

(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.

6.1.3 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 Processes for Priority scheduling

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

P 2	P_5	P_1	P_3	P 4

0 1 6

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.

6.1.4 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.

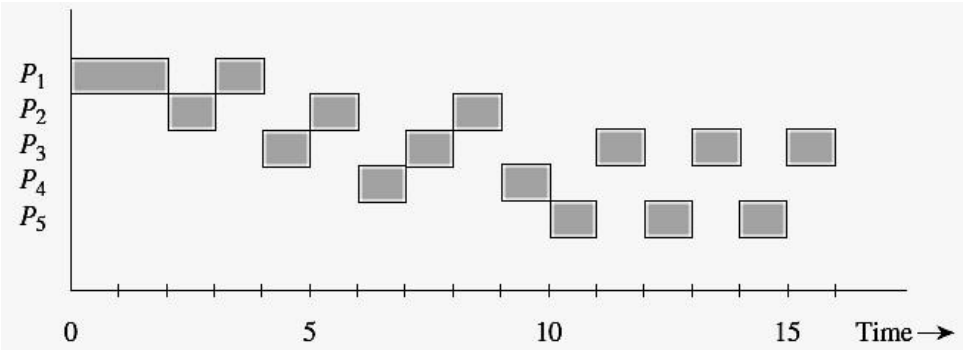
Figure RR scheduling summarizes operation of the RR scheduler with δ = 1 second for the five processes shown in Table. The scheduler makes scheduling decisions every second. The time when a decision is made is shown in the first row of the table in the top half of Figure 7.3. The next five rows show 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	c	t_a	w
Position of	P_1	1	1	2	1													4	4	1.33
Processes in	P_2			1	3	2	1	3	2	1								9	7	2.33
ready queue	P_3				2	1	3	2	1	4	3	2	1	2	1	2	1	16	13	2.60
(1 implies	P_4					3	2	1	3	2	1							10	6	3.00
head of queue)	P_5									3	2	1	2	1	2	1		15	7	2.33
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			

- **Table 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 Scheduling using Round-robin policy with timeslicing**

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 approximates

FCFS.

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

6.1.5 MULTILEVEL QUEUE SCHEDULING

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

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

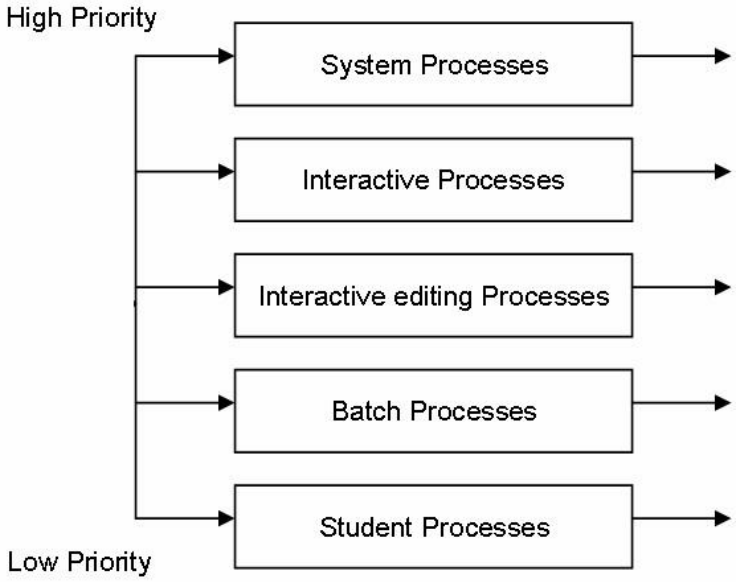


Fig Multilevel queue scheduling

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

The 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

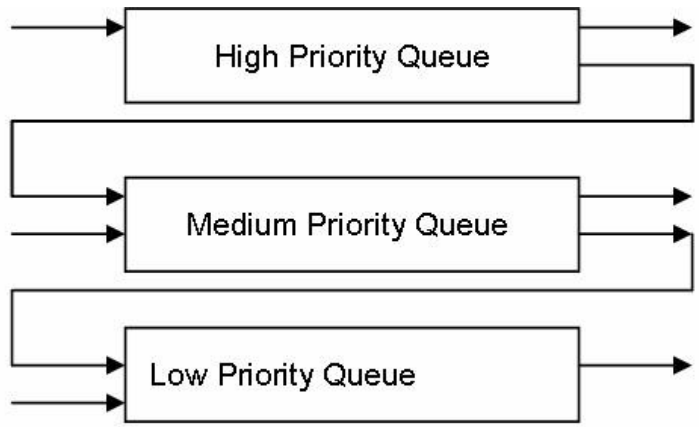
times, which it can then schedule among the processes in its queue. For instance;

- ✓ 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.

6.1.6 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**

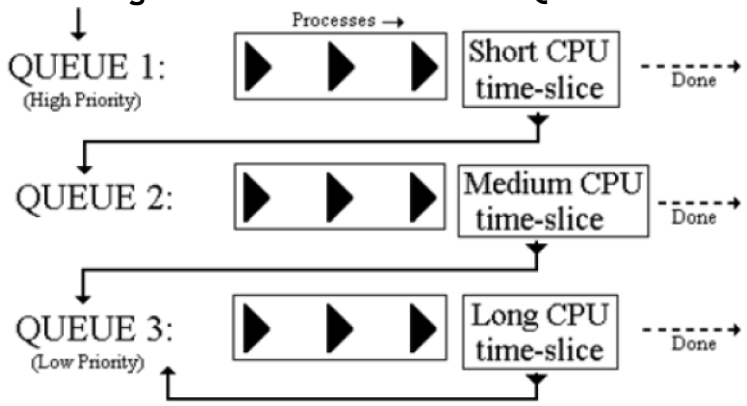


Fig MFQ Scheduling architecture

Multilevel feedback queue scheduler is characterized by the following parameters:

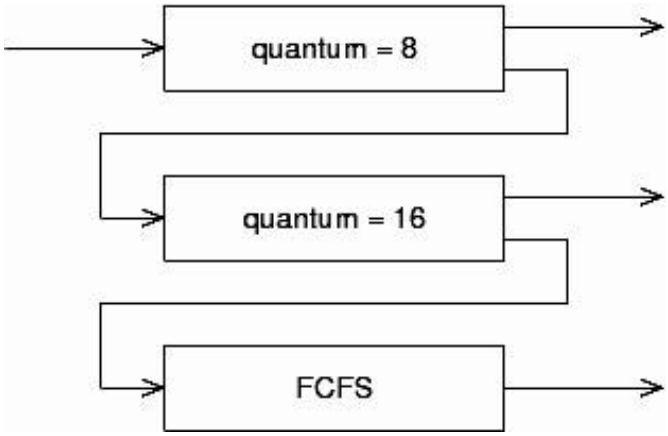
1. Number of queues
2. Scheduling g algorithms for each queue

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 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.

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.

7 Process and Thread Management

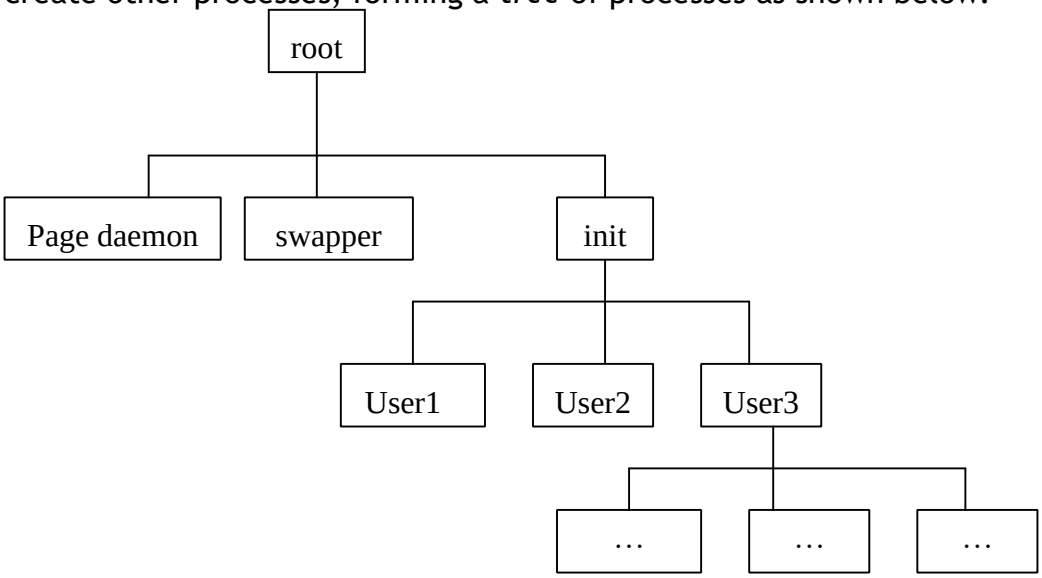
7.1 *INTRODUCTION*

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.

7.2 *OPERATIONS ON PROCESS*

7.2.1 PROCESS CREATION

A process may create several new processes, via a ***createprocess 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

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.

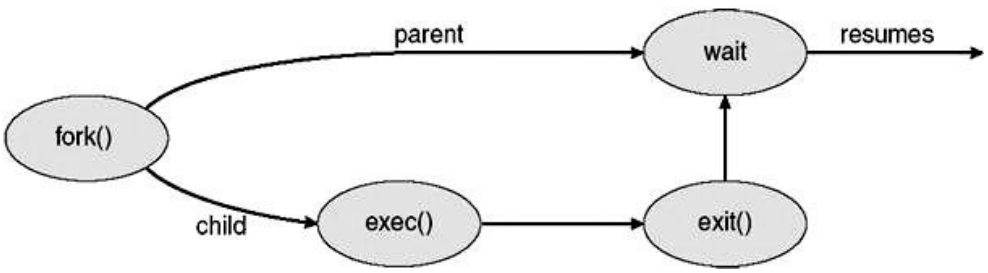
There are also two possibilities in terms of the address space of the new process:

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 UNIX Process Creation**

```
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.

7.2.2 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).

7.2.3 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.

7.3 *INTERPROCESS COMMUNICATION*

The OS provides the means for cooperating processes to communicate with each other via

an inter-process 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.

7.3.1 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.

7.3.1.1 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.

7.3.1.2 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 P2 or P3, 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.

7.3.1.3 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.

7.3.1.4 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.

7.4 *MULTITHREADING MODELS*

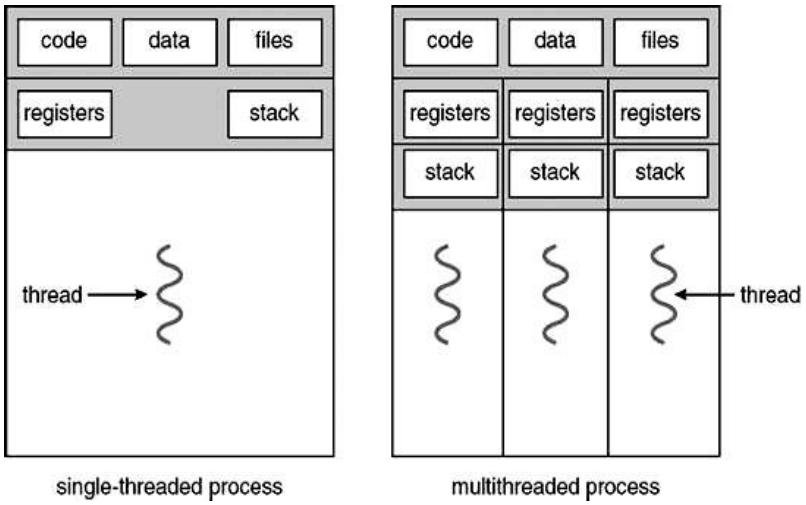


Fig Single threaded and multithreaded processes

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.

7.4.1 Many-to-One Model:

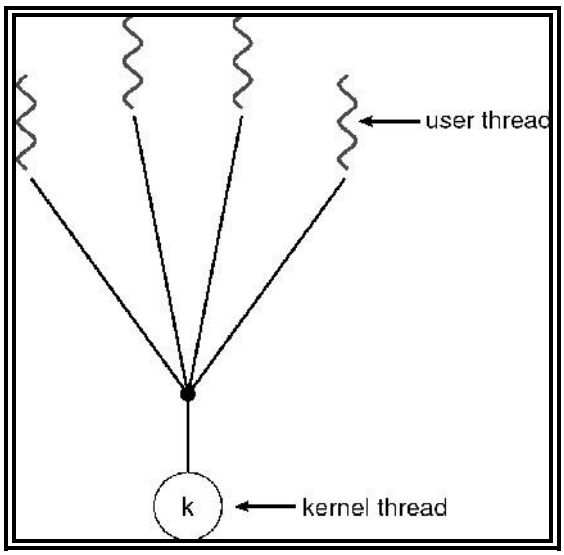
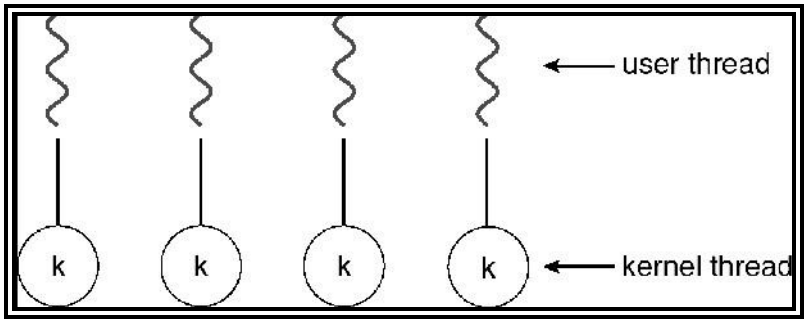


Fig 8.4 Many-to-One model

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.

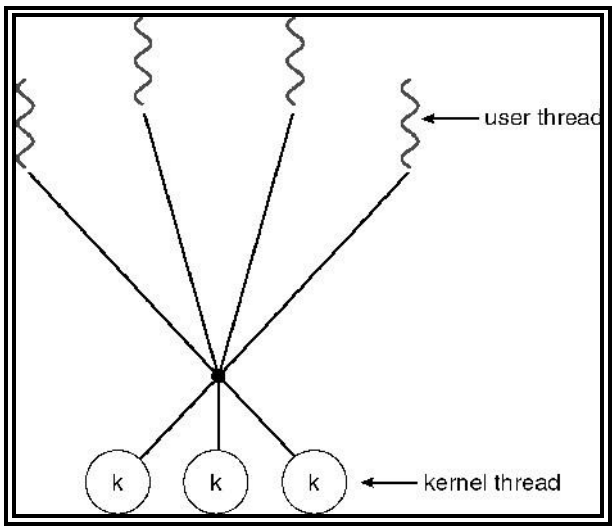
8.4.2 One-to-One Model:



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.

7.4.2 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.

7.5 *Thread features*

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.

7.5.1 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.

7.5.2 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 next Figure.

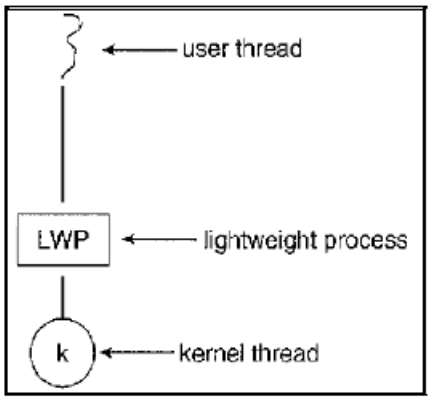


Fig 8.7 Lightweight Process (LWP)

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 CLIENT-SERVER SYSTEMS

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).

8.1 *COMMUNICATION IN CLIENT-SERVER SYSTEM*

Three strategies for communication in client-server systems are:

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

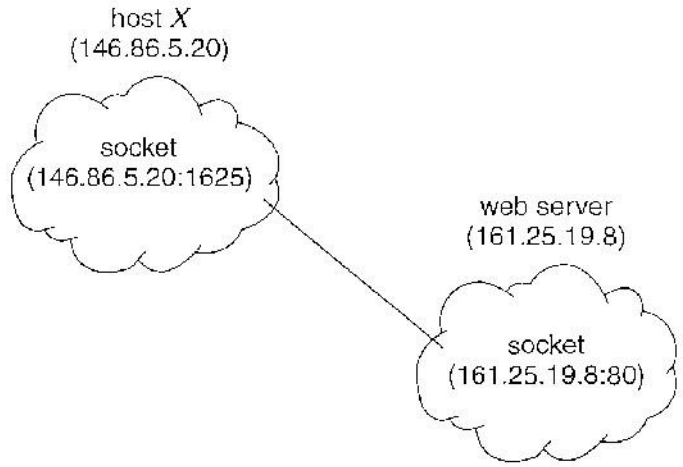
3. Pipes.

8.1.1 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 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 sockets 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 connection is received, the server returns the date and time to the client. The date server is shown in Program Figure.

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.

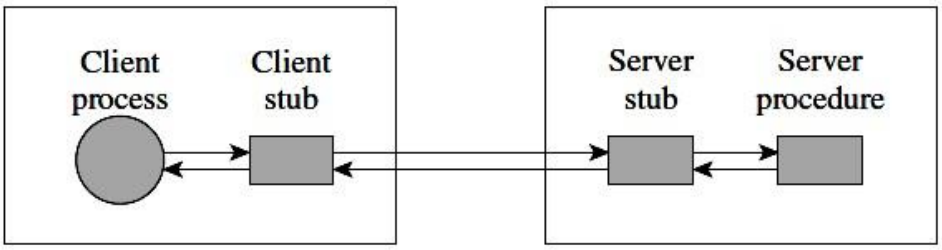
```
import java.net.*;
import java.io.*;
 public class DateClient{}

 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 Date client.

8.1.2 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.



Client node Server node
Fig 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 bigendian) 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.

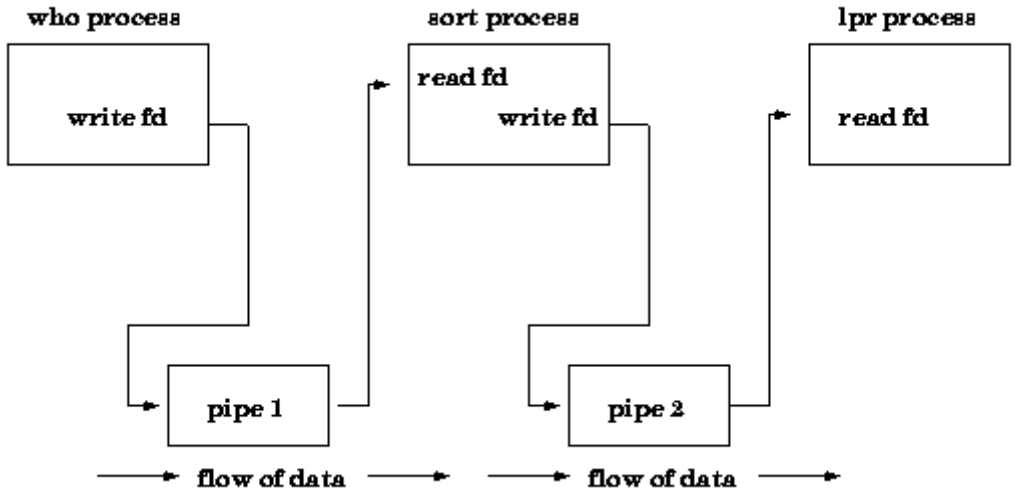
The RPC scheme is useful in implementing a distributed file system. Such a system can be implemented as a set of RPC daemons and clients. The messages are addressed to the distributed file system port on a server on which a file operation is to take place. The message contains the disk operation to be 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.

8.1.3 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.



Questions

1. Define : (a) Socket (b) Pipe
2. What is RPC?
3. Describe with a graph socket communication
4. Describe the structure of RPC

9 MAIN MEMORY

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 willreside 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 executionfrom one memory segment to another, then binding must be delayed until run time.

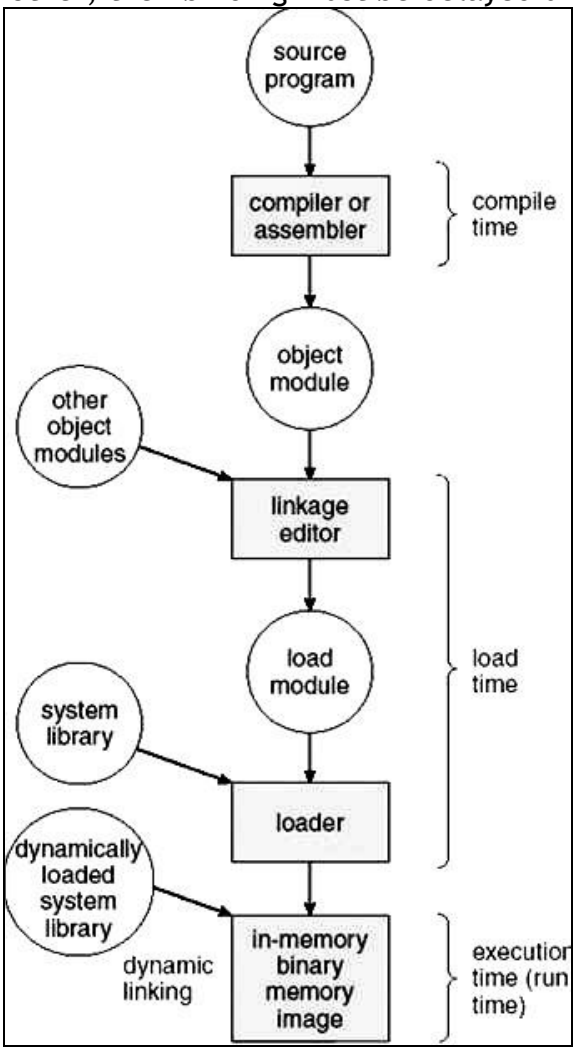


Fig BINDING OF INSTRUCTIONS

9.1 *MEMORY MANAGEMENT WITHOUT SWAPPING OR PAGING*

9.1.1 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.

9.1.2 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.

9.1.3 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

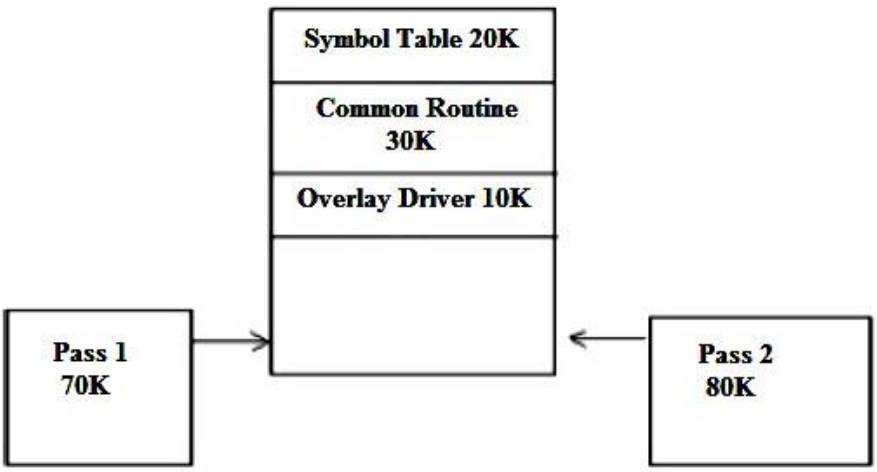


FIG OVERLAYS

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.

9.1.4 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.

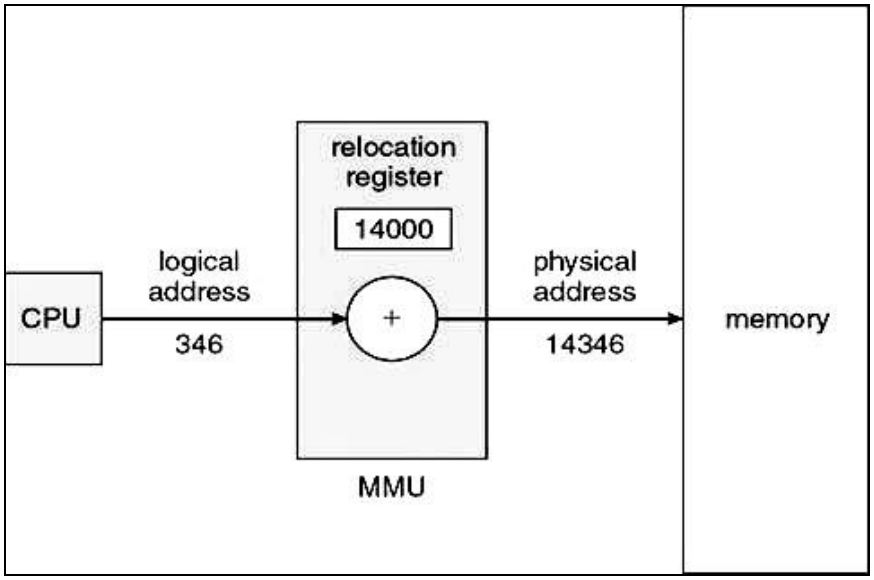


FIG DYNAMIC RELOCATION USING RELOCATION REGISTER

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.

9.2 *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.

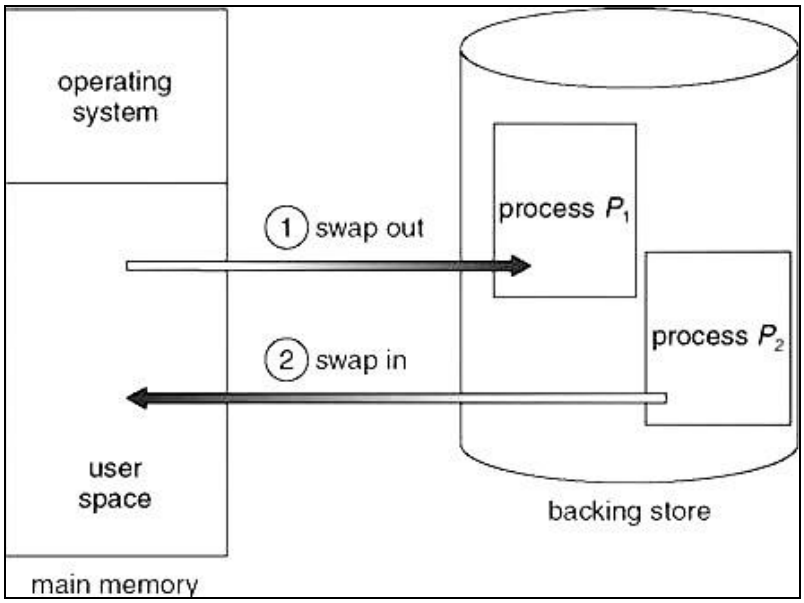


FIG SWAPPING OF TWO PROCESSES USING A DISK AND A BACKING STORE

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

9.3 *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.

9.3.1 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.

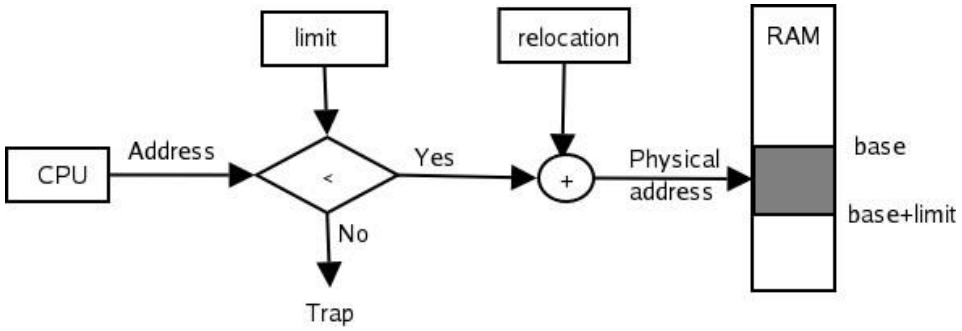
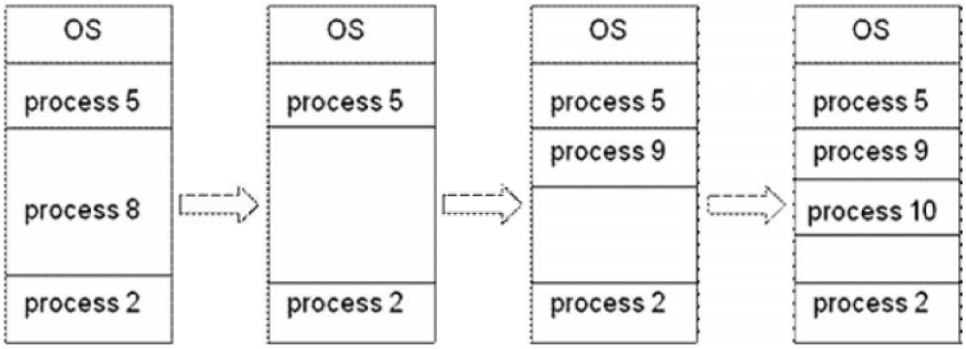


Fig HARDWARE SUPPORT FOR RELOCATION ANDLIMIT REGISTERS

9.3.2 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.



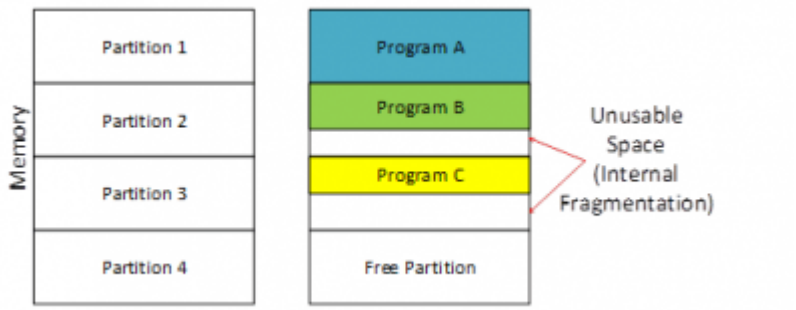
When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, 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.

9.3.3 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.



9.4 *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.

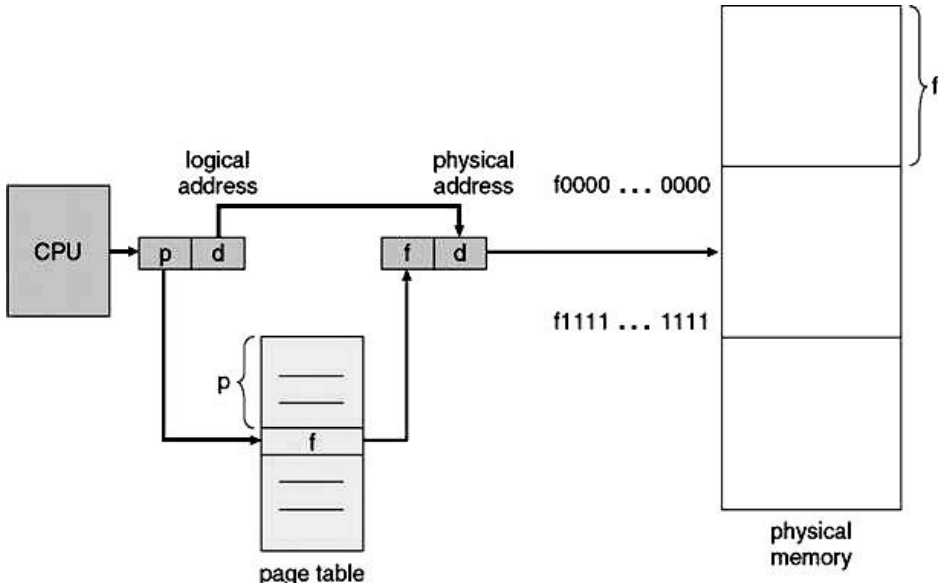


Fig Paging Hardware

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.

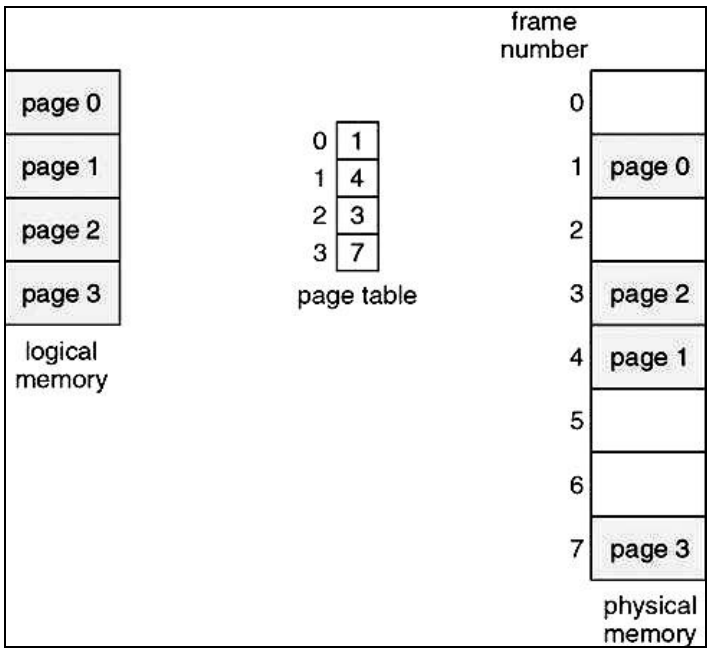


Fig Paging model of logical and physical memory

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.

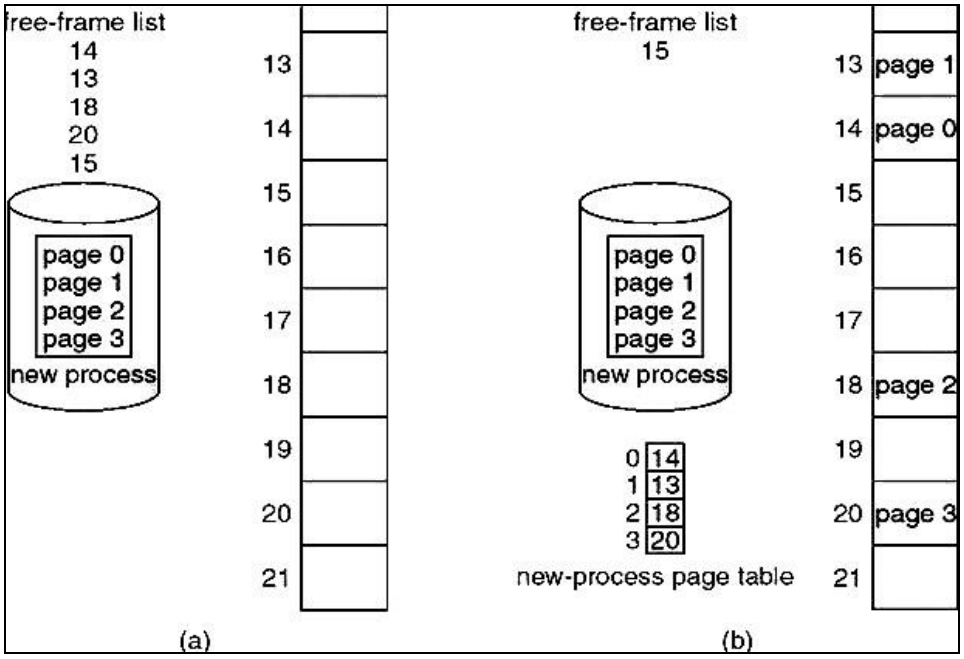


FIG FREE FRAMES (a)BEFORE ALLOCATION (b) AFTER ALLOCATION

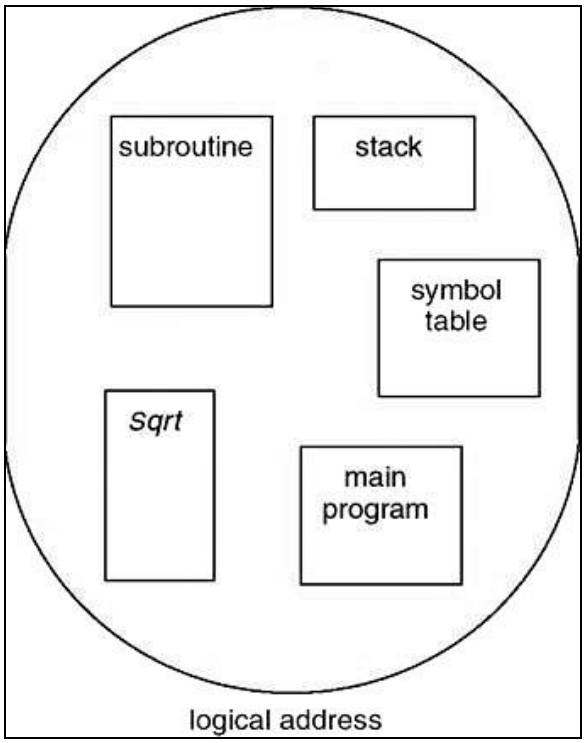
The user program views that memory as one single contiguous space, containing only this one program. But the user program is scattered throughout physical memory and logical addresses are translated into physical addresses.

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.

9.5 *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 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.

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

10 VIRTUAL MEMORY

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

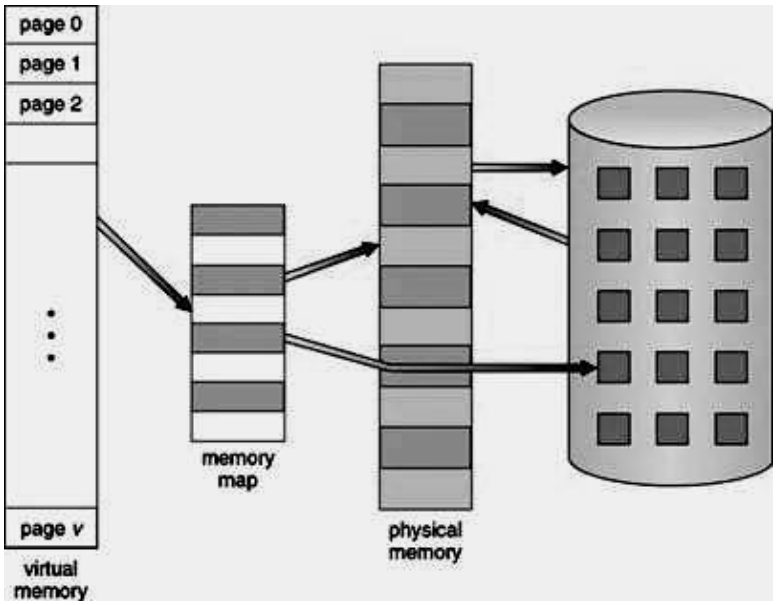


Fig Virtual memory is larger than physical memory

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.

10.1 *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.

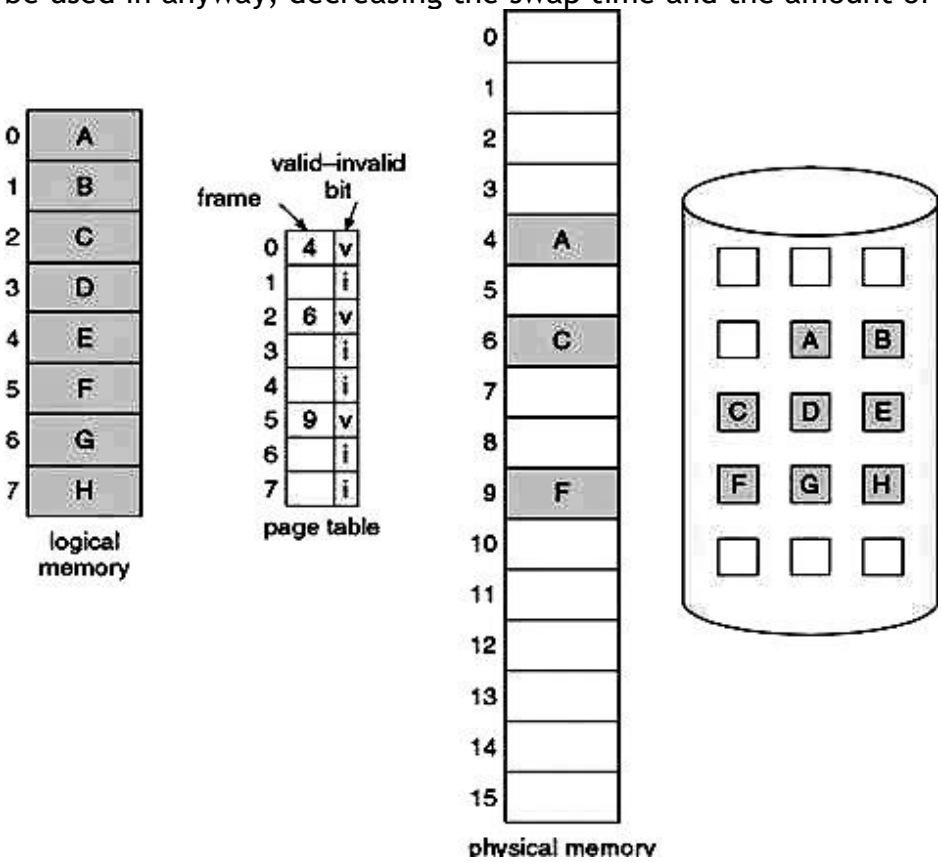


Fig Transfer of a paged memory to contiguous disk space

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:

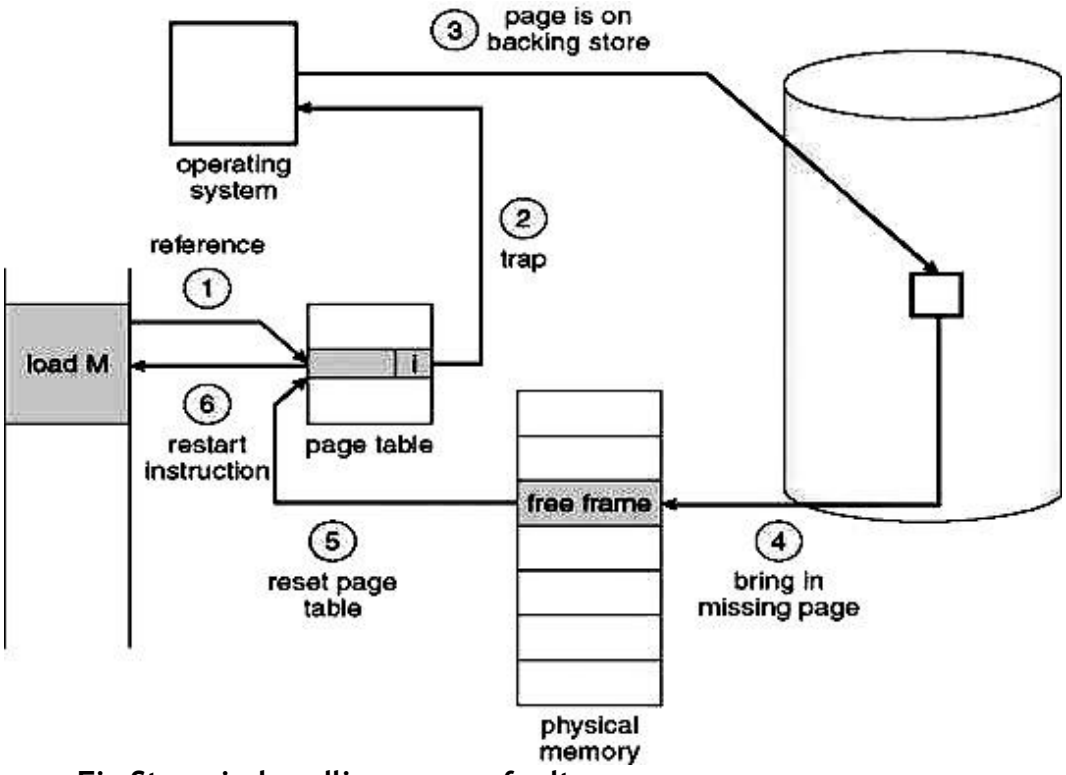


Fig Steps in handling a page fault

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.

10.2 *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.

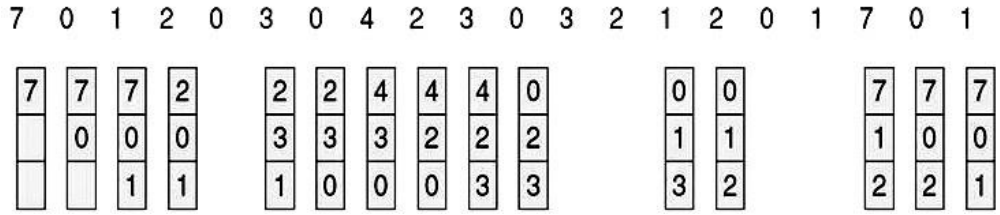
2. If we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

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.

10.2.1 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

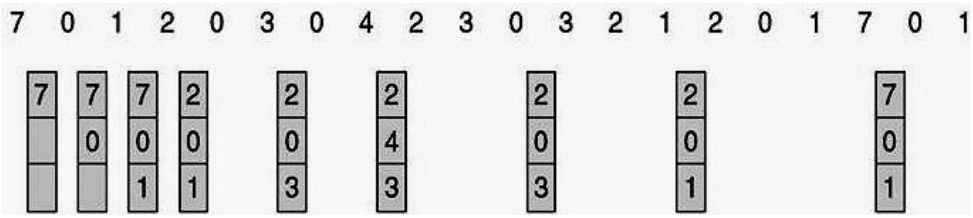


Page frames

- **Fig FIFO Page-replacement algorithm**

10.2.2 Optimal Algorithm

An optimal page-replacement algorithm has the lowest pagefault 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 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.

10.2.3 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

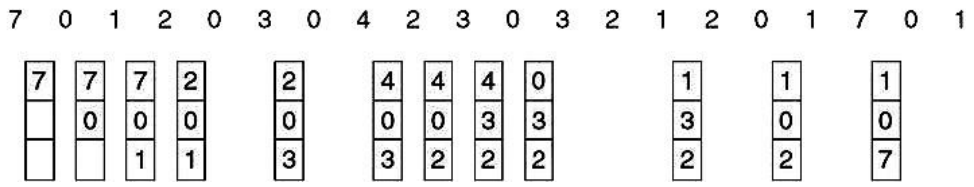


Fig LRU Algorithm

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.

10.2.4 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 modified, 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.

10.2.5 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.

10.3 *MODELING PAGING ALGORITHM*

10.3.1 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

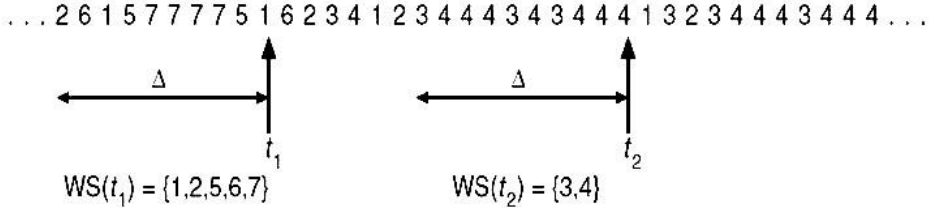


Fig Working-set model

- 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 workingset 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.

10.4 *DESIGN ISSUES FOR PAGING SYSTEMS*

10.4.1 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.

10.4.2 Page Size

Page size selection must take into consideration:

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

10.4.3 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

10.4.4 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.

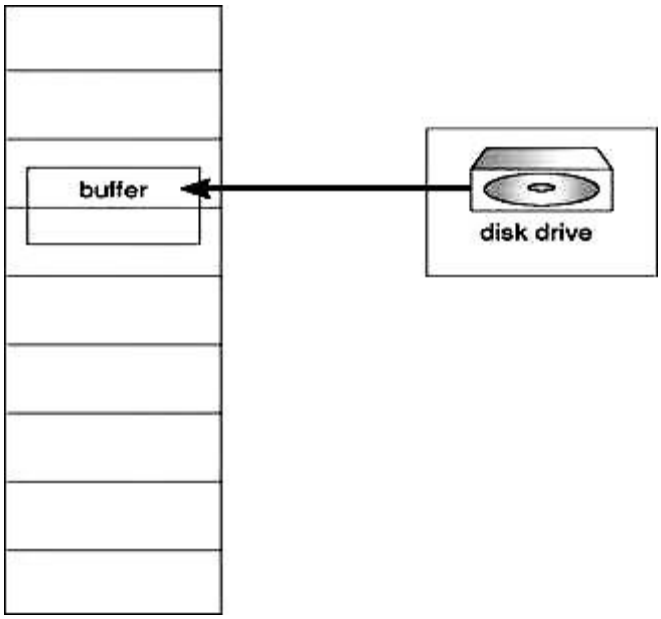


Fig The reason why frames used for 1/0 must be in memory

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.

11 FILE SYSTEM INTERFACE AND IMPLEMENTATION

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.

11.1 *FILE CONCEPTS*

11.1.1 FILE STRUCTURE

Three terms are used for files

- Field
- Record
- Database

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

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

A file is a collection of similar records. The file is treated as a single entity by users and applications and may be referenced by name. Files have file names and 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.

11.1.2 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.

11.1.3 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.

11.1.4 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.

11.2 *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.

11.2.1 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.

11.2.1.1 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.

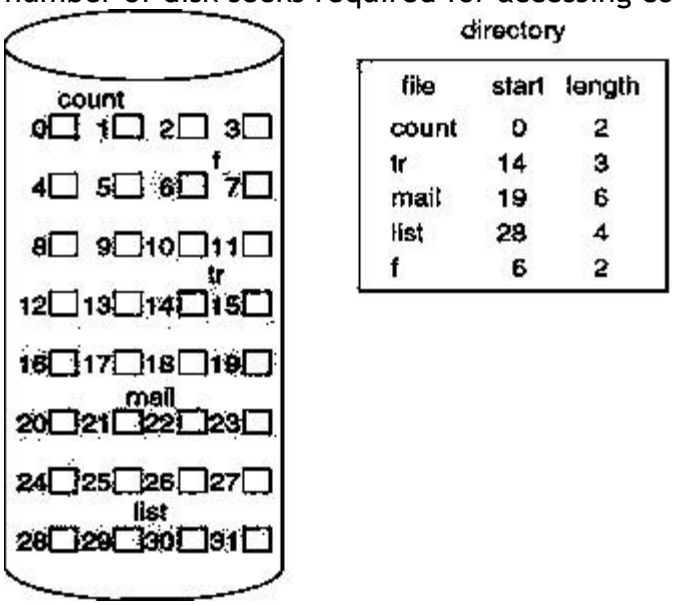


Fig Contiguous allocation of disk space

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 overestimate the amount of space needed, resulting in considerable wasted space.

11.2.1.2 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.

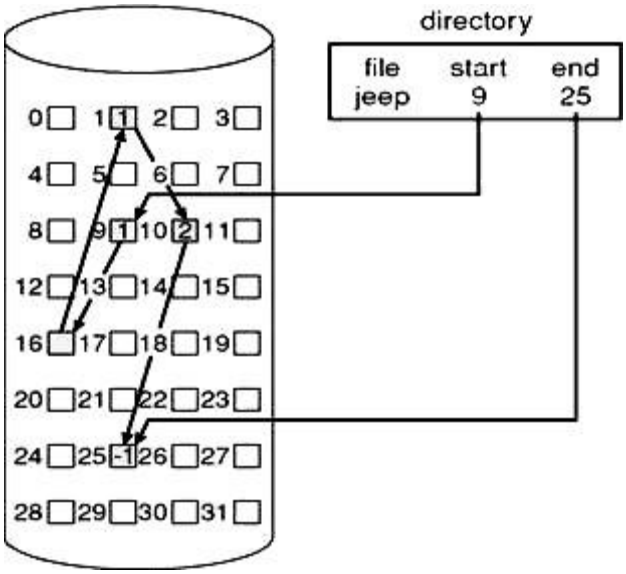


Fig Linked allocation of disk space

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 defines 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 lists or to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

An important variation, on the linked allocation method is the use of a file allocation table

(FAT). This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each partition is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value -as the table entry. Unused blocks are indicated by a 0 value in the table. 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.

11.2.1.3 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 i^{th} entry in the index block points to the i^{th} 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 i^{th} block is first written, a block is obtained: from the free space manager, and its address- is put in the i^{th} index block entry.

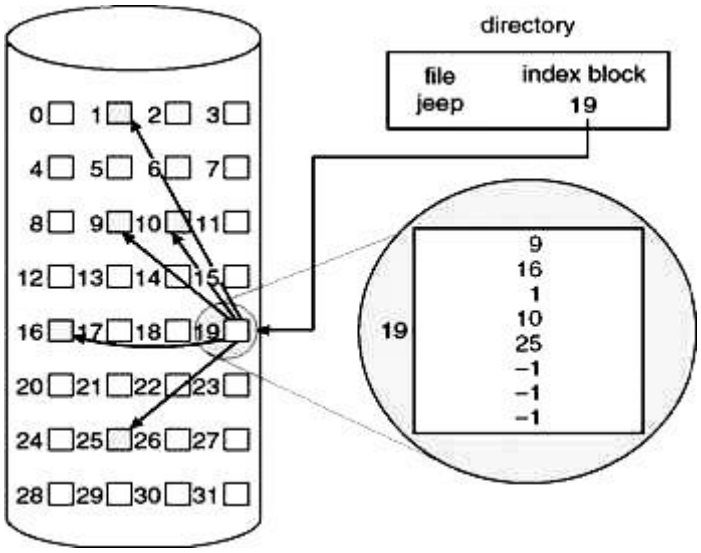


Fig 12.3 Indexed allocation of disk space

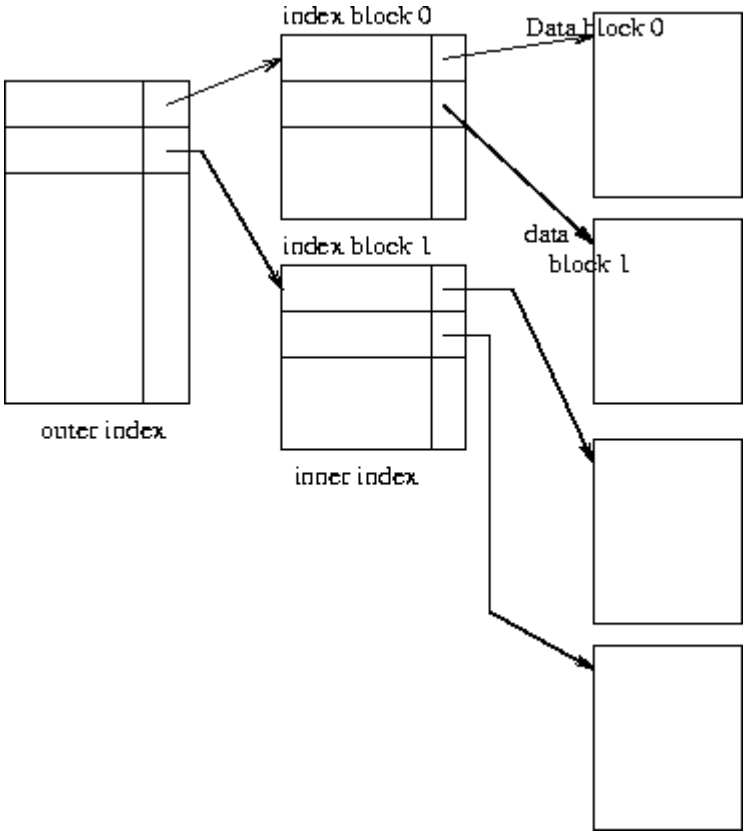
Allocation supports direct access, without suffering from external fragmentation because any free block on the disk may satisfy a request for more space. Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than

the pointer overhead of linked allocation.

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.



11.3 *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.

11.3.1 BIT VECTOR

The 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, considering 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

0011110011111100011000000011100000 .....

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

11.3.2 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, etc. 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.

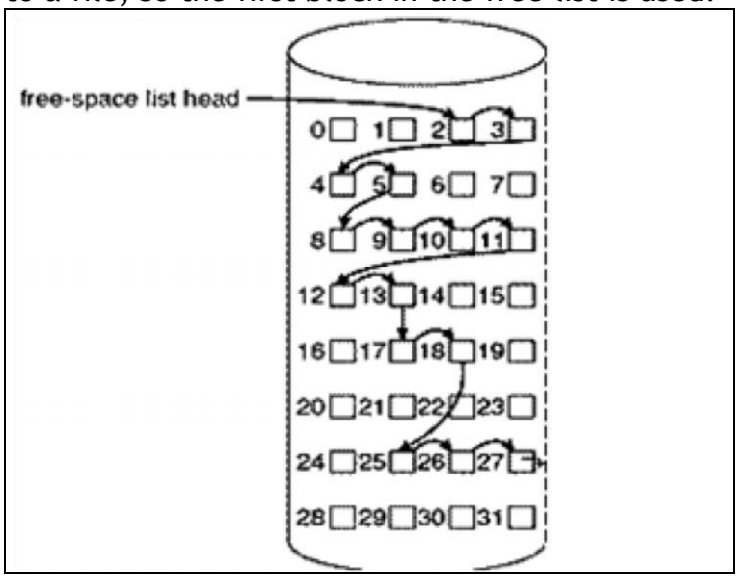


Fig 12.4 Linked free-space list on disk

11.3.3 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.

11.3.4 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.

11.4 *FILE SHARING*

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

11.4.1 MULTIPLE USERS

To implement sharing and protection, the system must maintain more file and directory attributes that 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.

11.4.2 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**

11.5 *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.

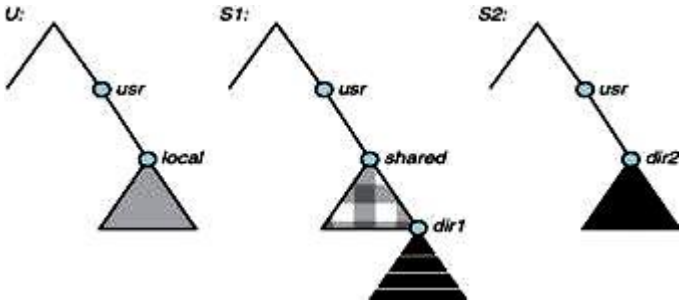


Fig Three Independent File Systems

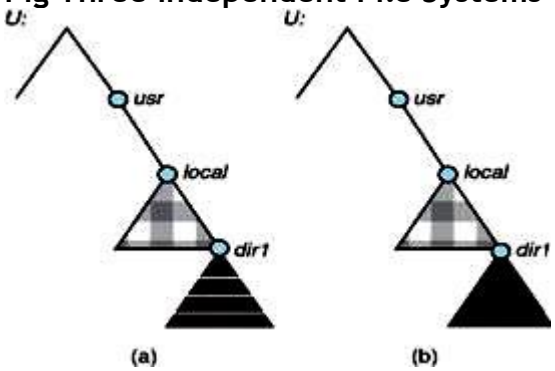


Fig Mounting in NFS
(a)Mounts (b) Cascading Mounts NFS Protocol

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 must 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.

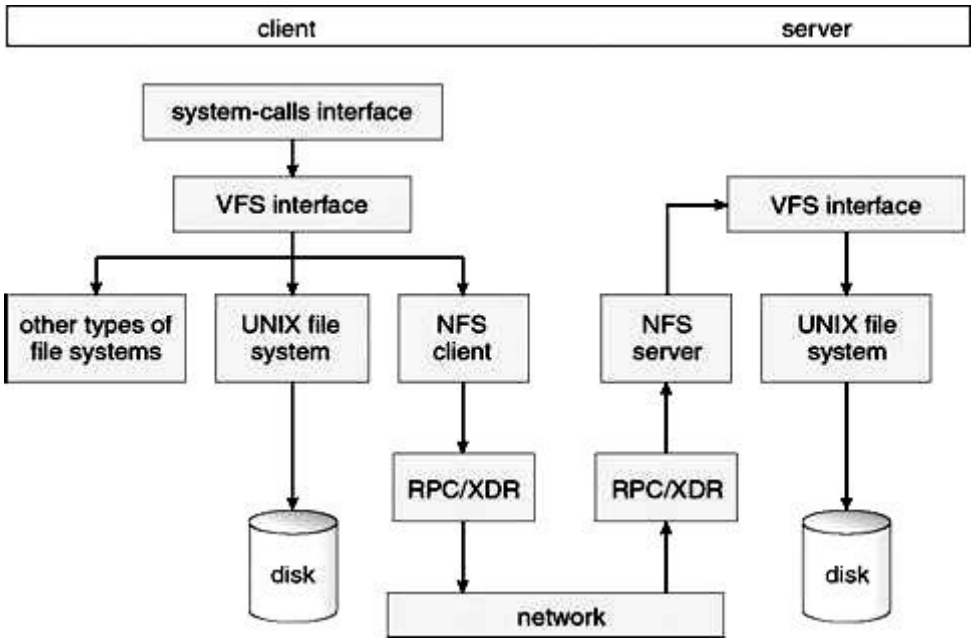


Fig Schematic View of NFS Architecture

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?

12 I/O SYSTEMS

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)

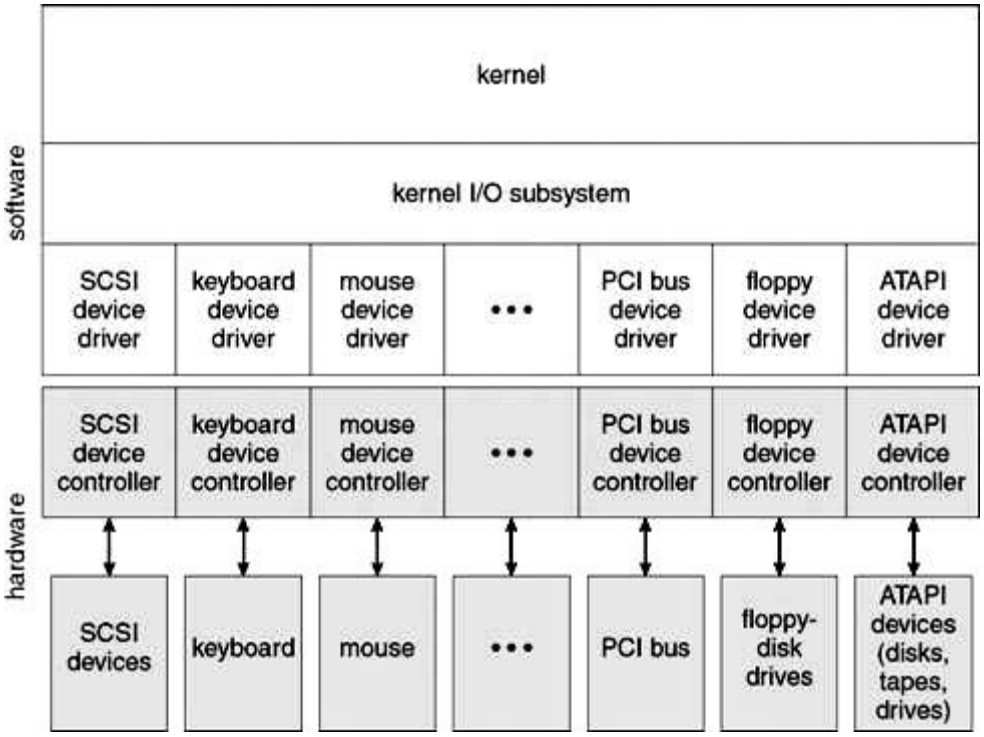
I/O Subsystems must contend with two trends:
(1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and
(2) The development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.

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

12.1 *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 Characteristics of I/O devices

12.1.1 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.

12.1.2 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.

12.1.3 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.

12.1.4 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)

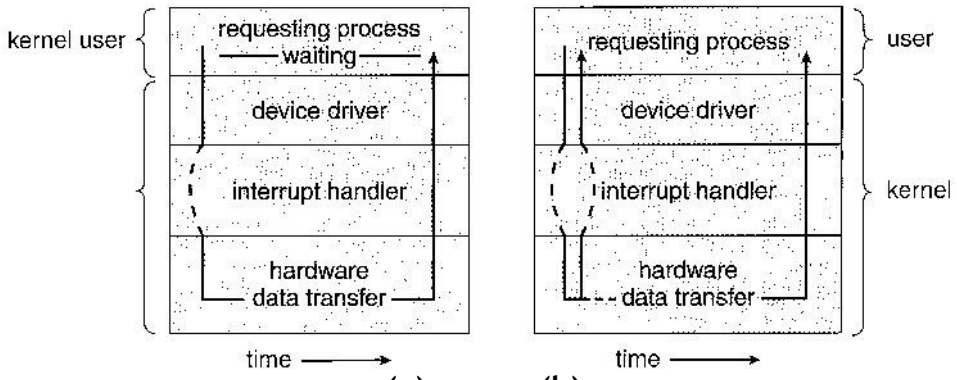


Fig Two I/O methods: (a) synchronous and

- **(b) asynchronous**

12.2 *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. Next figure illustrates the steps taken to process a (blocking) read request:

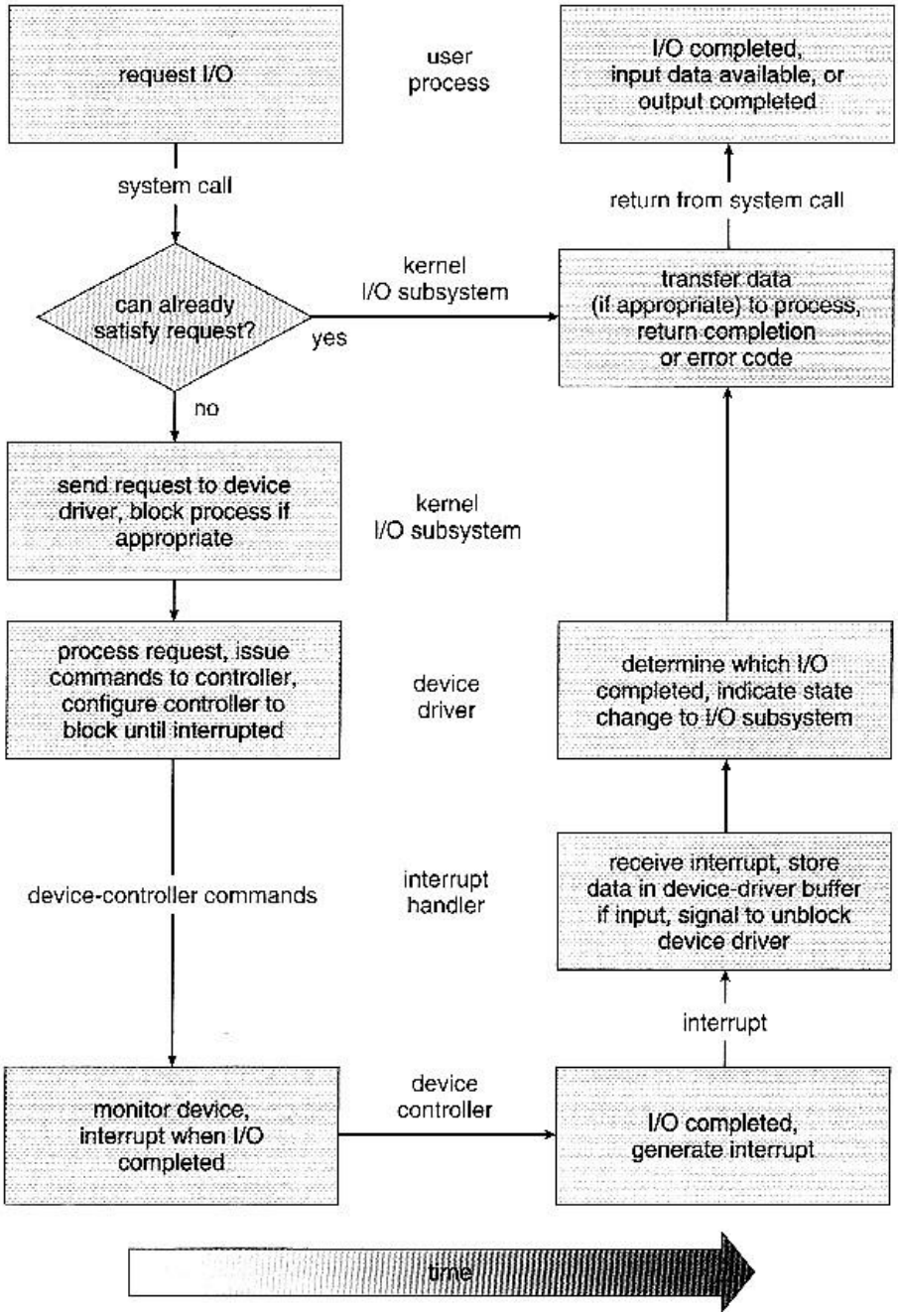


Fig. The life cycle of an I/O request

12.3 *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.

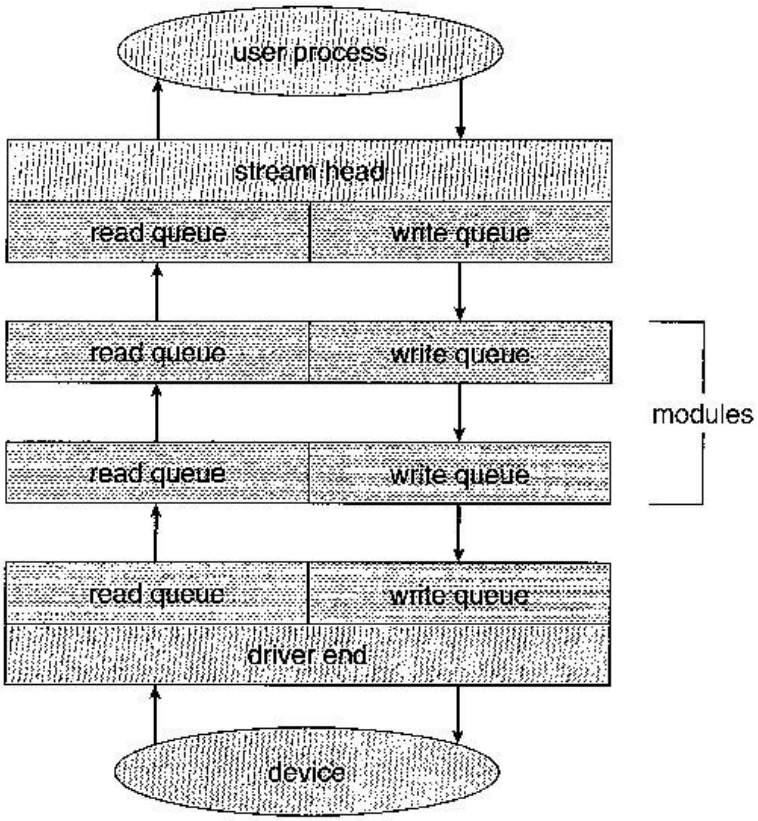


Fig. The STREAMS architecture

12.4 ***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.

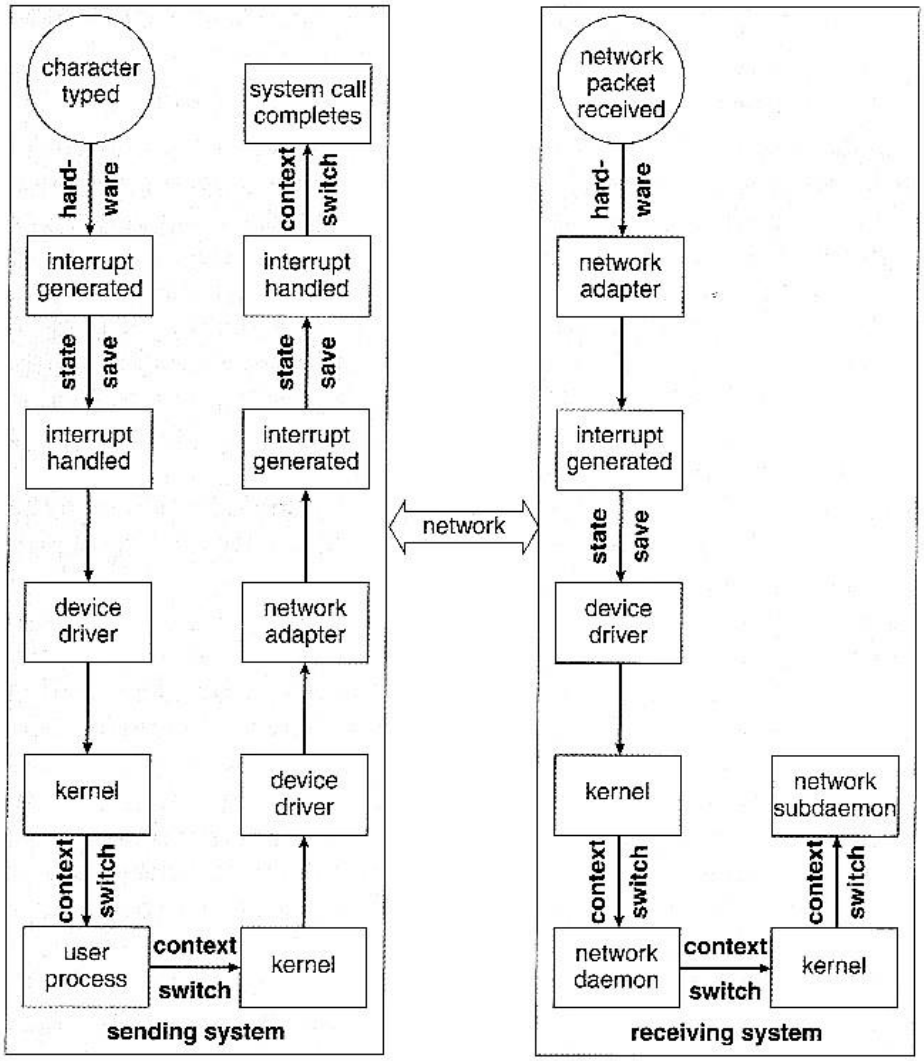


Fig Intercomputer Communications

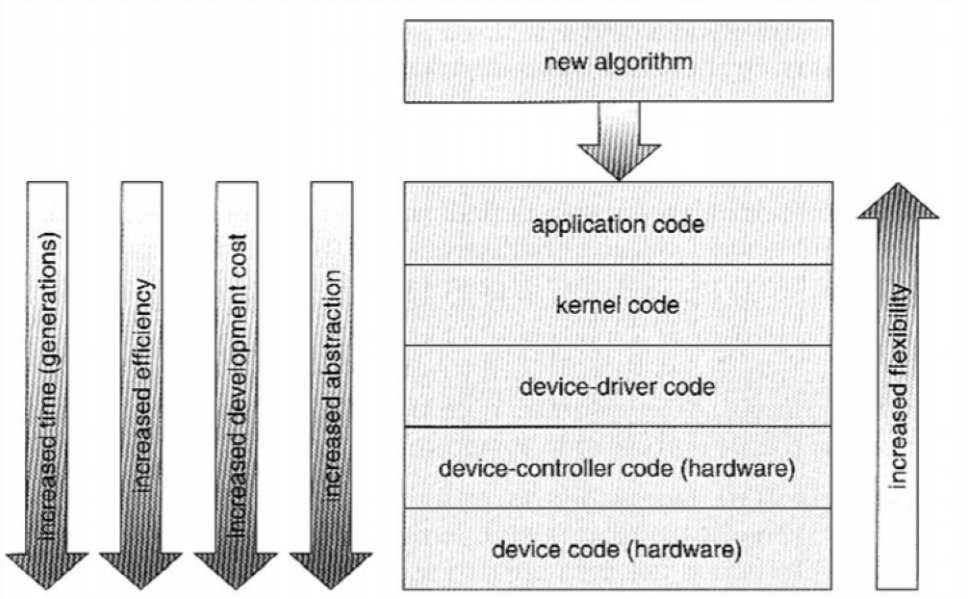
Other systems use front-end processors to off-load some of the work of I/O processing from the CPU. For example a terminal concentrator can multiplex with hundreds of terminals on a single port on a large computer.

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**

Questions

1) State some characteristics of I/O devices.
2) Write a short note on Blocking and non-blocking I/O
3) Describe the life cycle of an I/O request with the help of a diagram.
4) 10.What are STREAMS?
5) 11.State various principles employed to increase the overall efficiency of I/O processing.

Unit 7. Operative systems