DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DIGITAL NOTES

ON

OPERATING SYSTEMS

R22A0509

B.TECH II YEAR–II SEM

(R22) REGULATION

(2024-25)

Prepared by

K SHANTHI

MALLAREDDY COLLEGE OF ENGINEERING &TECHNOLOGY
(Autonomous Institution–UGC,Govt.of India)
Recognized under2(f)and12(B) ofUGC ACT1956
(Affiliated to JNTUH,Hyderabad,ApprovedbyAICTE-Accredited by NBA&NAAC–'A'Grade-ISO 9001:2015 Certified)
Maisammaguda,Dhulapally(PostVia.Hakimpet),Secunderabad–500100,TelanganaState,India

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

II Year B.Tech. CSE- II Sem **L/T/P/C**
3/-/-/3

(R22A0509)

OPERATING SYSTEMS

Course Objectives:

1. To understand the fundamental concepts and techniques of Operating Systems.
2. To studythe concepts of linux shell and scheduling.
3. To understand the concepts in deadlocks and process management.
4. To understand the concepts in memory managements and IPC mechanism.
5. To studythe concept of file system concepts and sockets.

UNIT – I

Operating System-Introduction, Structures-Simple Batch, Multi-programmed, Time-shared, Personal Computer, Parallel, Distributed Systems, Real-Time Systems , System components, Operating System services.

Introduction to Linux operating system, linux file system, Linux Utilities

UNIT – II

Linux: Introduction to shell, Types of shell's , example shell programs.

Process and CPU Scheduling - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads, Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling.

UNIT – III

Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock

Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors

UNIT – IV

Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory implementation in linux. Corresponding system calls.

Memory Management and Virtual Memory - Logical versus Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation, Segmentation with Paging, Demand Paging, Page Replacement, Page Replacement Algorithms.

UNIT - V

File System Interface and Operations -Access methods, Directory Structure, Protection, File System Structure, Allocation methods, kernel support for files, system calls for file I/O operations open, create, read, write, close, lseek, stat, ioctl Disk Management: Disk Scheduling Algorithms-FCFS, SSTF, SCAN, C-SCAN

TEXT BOOKS:

1. Beginning Linux Programming –Neil Mathew,Richard Stones 4th Edition,Wiley
2. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7 th Edition,John Wiley
3. Unix System Programming using C++, T.Chan,PHI.
4. Unix Concepts and Applications, 4th Edition, SumitabhaDas,TMH,2006.
5. Advanced programming in the UNIX environment, W.R. Stevens, Pearson education.

REFERENCE BOOKS:

1. Operating Systems – Internals and Design Principles Stallings, Fifth Edition–2005, Pearson Education/PHI
2. Operating System A Design Approach- Crowley, TMH.
3. Modern Operating Systems, Andrew S. Tanenbaum 2 nd edition, Pearson/PHI
4. UNIX programming environment, Kernighan and Pike, PHI/ Pearson Education
5. UNIX Internals -The New Frontiers, U. Vahalia, Pearson Education.

Course Outcomes:

1. An ability to understand basic concepts of operating system.
2. An ability to understand linux shell and scheduling.
3. An ability to analyze memory management and deadlocks.
4. An ability to describe memory management and concurrency control mechanisms.
5. An ability to compare various file systems

INDEX

UNIT-I

Operating System-Introduction, Structures-Simple Batch, Multi-programmed, Time-shared, Personal Computer,Parallel, Distributed Systems, Real-Time Systems , System components, Operating System services.

Introduction to Linux operating system, linux file system, Linux Utilities

A computer system has many resources (hardware and software), which may be required to complete a task. The commonly required resources are input/output devices, memory, file storage space, CPU, etc. The operating system acts as a manager of the above resources and allocates them to specific programs and users, whenever necessary to perform a particular task. Therefore the operating system is the resource manager i.e. it can manage the resource of a computer system internally. The resources are processor, memory, files, and I/O devices.

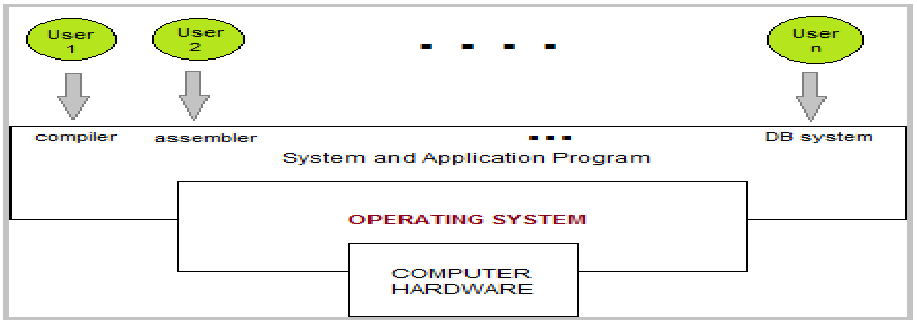
In simple terms, an operating system is an interface between the computer user and the machine.

It is very important for you that every computer must have an operating system in order to run other programs. The operating system mainly coordinates the use of the hardware among the various system programs and application programs for various users.

An operating system acts similarly like government means an operating system performs no useful function by itself; though it provides an environment within which other programs can do useful work.

Below we have an abstract view of the components of the computer system:

Four Components of a Computer System



In the above picture:

- The **Computer Hardware** contains a central processing unit(CPU), the memory, and the input/output (I/O) devices and it provides the basic computing resources for the system.

- The **Application programs like spreadsheets, Web browsers, word processors, etc.** are used to define the ways in which these resources are used to solve the computing problems of the users. And the System program mainly consists of compilers, loaders, editors, OS, etc.

- The Operating System is mainly used to control the hardware and coordinate its use among the various application programs for the different users.
- Basically, Computer System mainly consists of hardware, software, and data.

OS is mainly designed in order to serve two basic purposes:

1. The operating system mainly controls the allocation and use of the computing System's resources among the various user and tasks.
2. It mainly provides an interface between the computer hardware and the programmer that simplifies and makes feasible for coding, creation of application programs and debugging

Two Views of Operating System

1. User's View
2. System View

Operating System: User View

The user view of the computer refers to the interface being used. Such systems are designed for one user to monopolize its resources, to maximize the work that the user is performing. In these cases, the operating system is designed mostly for ease of use, with some attention paid to performance, and none paid to resource utilization.

Operating System: System View

The operating system can be viewed as a resource allocator also. A computer system consists of many resources like - hardware and software - that must be managed efficiently. The operating system acts as the manager of the resources, decides between conflicting requests, controls the execution of programs, etc.

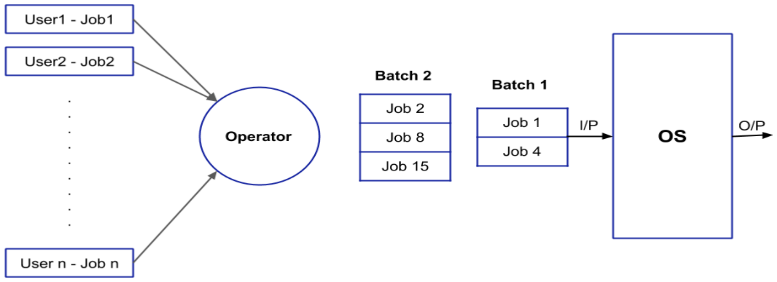
Types of Operating System

Given below are different types of Operating System:

1. Simple Batch System
2. Multiprogrammed
3. Time-Shared
4. Personal Computer
5. Parallel
6. Distributed Systems
7. Real Time Systems

1. Simple Batch System

In a Batch Operating System, the similar jobs are grouped together into batches with the help of some operator and these batches are executed one by one. For example, let us assume that we have 10 programs that need to be executed. Some programs are written in C++, some in C and rest in Java. Now, every time when we run these programmes individually then we will have to load the compiler of that particular language and then execute the code. But what if we make a batch of these 10 programmes. The benefit with this approach is that, for the C++ batch, you need to load the compiler only once. Similarly, for Java and C, the compiler needs to be loaded only once and the whole batch gets executed. The following image describes the working of a Batch Operating System.



Advantages:

1. The overall time taken by the system to execute all the programmes will be reduced.
2. The Batch Operating System can be shared between multiple users.

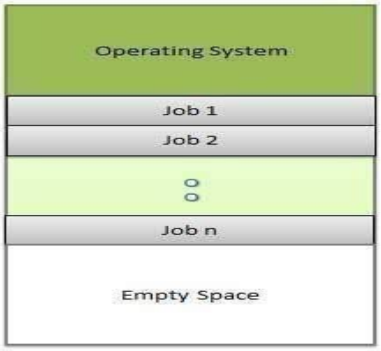
Disadvantages:

1. Manual interventions are required between two batches.
2. The CPU utilization is low because the time taken in loading and unloading of batches is very high as compared to execution time.

Multiprogramming

Sharing the processor, when two or more programs reside in memory at the same time, is referred as **multiprogramming**. Multiprogramming assumes a single shared processor. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

The following figure shows the memory layout for a multiprogramming



system.

An OS does the following activities related to multiprogramming.

- The operating system keeps several jobs in memory at a time.
- This set of jobs is a subset of the jobs kept in the job pool.
- The operating system picks and begins to execute one ofthe jobs in the memory.
- Multiprogramming operating systems monitor the state of all active programs and system resources using memory management programs to ensures that the CPU is never idle, unless there are no jobs to process.

Advantages

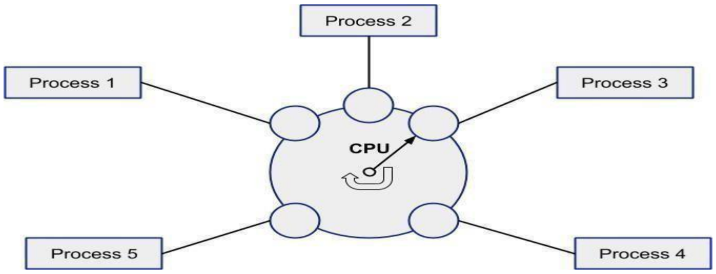
- High and efficient CPU utilization.
- User feels that many programs are allotted CPU almost simultaneously.

Disadvantages

- CPU scheduling is required.
- To accommodate many jobs in memory, memory management is required.

Time-Sharing Operating System

In a Multi-tasking Operating System, more than one processes are being executed at a particular time with the help of the time-sharing concept. So, in the time-sharing environment, we decide a time that is called time quantum and when the process starts its execution then the execution continues for only that amount of time and after that, other processes will be given chance for that amount of time only. In the next cycle, the first process will again come for its execution and it will be executed for that time quantum only and again next process will come. This process will continue. The following image describes the working of a Time-Sharing Operating System.



Advantages:

1. Since equal time quantum is given to each process, so each process gets equal opportunity to execute.
2. The CPU will be busy in most of the cases and this is good to have case.

Disadvantages:

1. Process having higher priority will not get the chance to be executed first because the equal opportunity is given to each process.

Personal Computers

Personal computer operating system provides a good interface to a single user.

Personal computer operating systems are widely used for word processing, spreadsheets and Internet access.

Personal computer operating system are made only for personal.You can say that your laptops, computer systems, tablets etc. are your personal computers and the operating system such as windows 7, windows 10, android, etc. are your personal computer operating system.

And you can use your personal computer operating system for your personal purposes, for example, to chatting with your friends using some social media sites, reading some articles from internet, making some projects through microsoft powerpoint or any other, designing your website, programming something, watching some videos and movies, listening to some songs and many more.

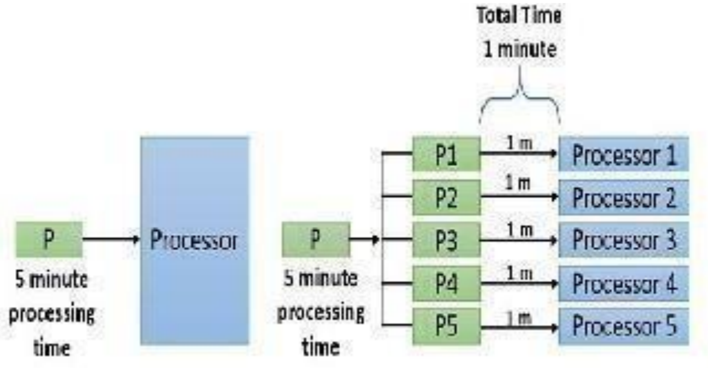
Applications
APP
APP
APP
APP
Monitor
Printer
Hard Drive
Operating System
Mouse
Keyboard

Parallel Processing

Parallel processing **requires multiple processors and all the processor works simultaneously in the system. Here, the task is divided into subparts and these subparts are then distributed among the available processors in the system. Parallel processing completes the job on the shortest possible time.**

All the processors in the parallel processing environment should run on the **same operating system**. All processors here are **tightly coupled** and are packed in one casing. All the processors in the system share the common **secondary storage** like the hard disk. As this is the first place where the programs are to be placed.

There is one more thing that all the processors in the system share i.e. the **user terminal** (from where the user interact with the system). The user need not to be aware of the inner architecture of the machine. He should feel that he is dealing with the single processor only and his interaction with the system would be the same as in a single processor,



Single Processor Vs Multiprocessor in Parallel processing

Advantages

1. It saves time and money as many resources working together will reduce the time and cut potential costs.

2. It can be impractical to solve larger problems on Serial Computing.

3. It can take advantage of non-local resources when the local resources are finite.

4. Serial Computing 'wastes' the potential computing power, thus Parallel Computing makes better work of the hardware.

Disadvantages

1. It addresses such as communication and synchronization between multiple sub-tasks and processes which is difficult to achieve.

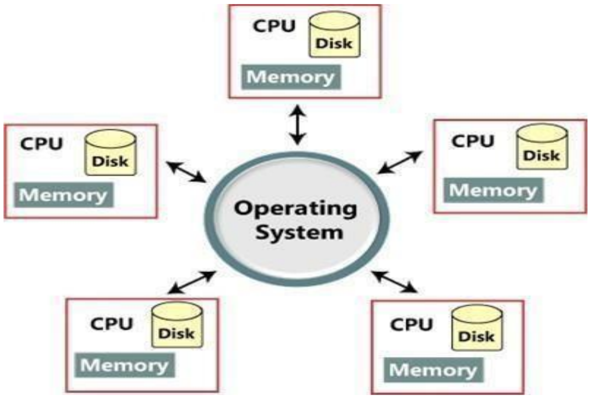
2. The algorithms must be managed in such a way that they can be handled in a parallel mechanism.

3. The algorithms or programs must have low coupling and high cohesion. But it's difficult to create such programs.

4. More technically skilled and expert programmers can code a parallelism-based program well.

Distributed Operating System

These types of the operating system is a recent advancement in the world of computer technology and are being widely accepted all over the world and, that too, with a great pace. Various autonomous interconnected computers communicate with each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred to as **loosely coupled systems** or distributed systems. These system's processors differ in size and function. The major benefit of working with these types of the operating system is that it is always possible that one user can access the files or software which are not actually present on his system but some other system connected within this network i.e., remote access is enabled within the devices connected in that network.



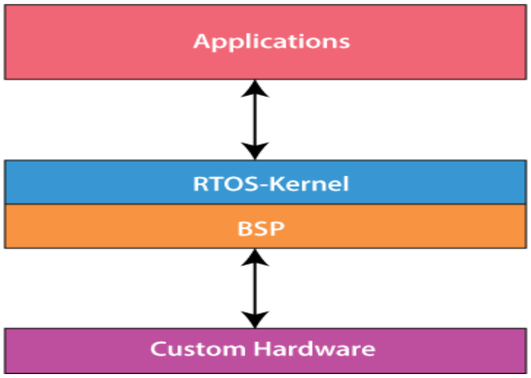
Advantages of Distributed Operating System:

- Failure of one will not affect the other network communication, as all systems are independent from each other
- Electronic mail increases the data exchange speed
- Since resources are being shared, computation is highly fast and durable
- Load on host computer reduces
- These systems are easily scalable as many systems can be easily added to the network
- Delay in data processing reduces

Disadvantages of Distributed Operating System:

- Failure of the main network will stop the entire communication
- To establish distributed systems the language which is used are not well defined yet
- These types of systems are not readily available as they are very expensive.

Real-Time Operating System:



It is developed for real-time applications where data should be processed in a fixed, small duration of time. It is used in an environment where multiple processes are supposed to be accepted and processed in a short time. RTOS requires quick input and immediate response, e.g., in a petroleum refinery, if the temperate gets too high and crosses the threshold value, there should be an immediate response to this situation to avoid the explosion. Similarly, this system is used to control scientific instruments, missile launch systems, traffic lights control systems, air traffic control systems, etc.

This system is further divided into two types based on the time constraints:

Hard Real-Time Systems:

These are used for the applications where timing is critical or response time is a major factor; even a delay of a fraction of the second can result in a disaster. For example, airbags and automatic parachutes that open instantly in case of an accident. Besides this, these systems lack virtual memory.

Soft Real-Time Systems:

These are used for application where timing or response time is less critical. Here, the failure to meet the deadline may result in a degraded performance instead of a disaster. For example, video surveillance (cctv), video player, virtual reality, etc. Here, the deadlines are not critical for every task every time.

Advantages of real-time operating system:

- The output is more and quick owing to the maximum utilization of devices and system
- Task shifting is very quick, e.g., 3 microseconds, due to which it seems that several tasks are executed simultaneously
- Gives more importance to the currently running applications than the queued application
- It can be used in embedded systems like in transport and others.
- It is free of errors.
- Memory is allocated appropriately.

Disadvantages of real-time operating system:

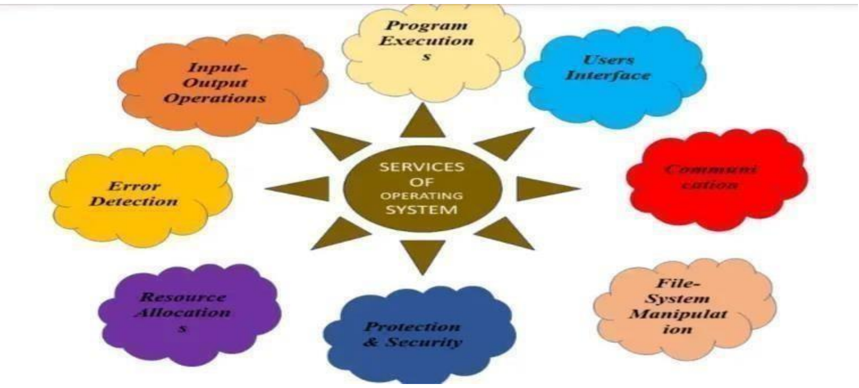
- A fewer number of tasks can run simultaneously to avoid errors.
- It is not easy for a designer to write complex and difficult algorithms or proficient programs required to get the desired output.
- Specific drivers and interrupt signals are required to respond to interrupts quickly.
- It may be very expensive due to the involvement of the resources required to work.

An operating system is an interface which provides services to both the user and to the programs. It provides an environment for the program to execute. It also provides users with the services of how to execute programs in a convenient manner. The operating system provides some services to program and also to the users of those programs. The specific services provided by the OS are off course different.

Following are the common services provided by an operating system:

1. Program execution

2. I/O operations
3. File system manipulation
4. Communication
5. Error detection
6. Resource allocation
7. Protection



1) Program Execution

- An operating system must be able to load many kinds of activities into the memory and to run it.

The program must be able to end its execution, either normally or abnormally.

- A process includes the complete execution of the written program or code. There are some of the

activities which are performed by the operating system:

- The operating system Loads program into memory
- It also Executes the program
- It Handles the program’s execution
- It Provides a mechanism for process synchronization
- It Provides a mechanism for process communication

2) I/O Operations

- The communication between the user and devices drivers are managed by the operating system.
- I/O devices are required for any running process. In I/O a file or an I/O devices can be involved.
- I/O operations are the read or write operations which are done with the help of input-output

devices.

- Operating system give the access to the I/O devices when it required.

3) File system manipulation

- The collection ofrelated information which represent some content is known as a file. The computer can

store files on the secondary storage devices. For long-term storage purpose. examples of storage media include magnetic tape, magnetic disk and optical disk drives like CD, DVD.

- A file system is a collection of directories for easy understand and usage. These directories contain some files. There are some major activities which are performed by an operating system with respect to file management.

○ The operating system gives an access to the program for performing an operation on the file.

○ Programs need to read and write a file.

○ The user can create/delete a file by using an interface provided by the operating system.

○ The operating system provides an interface to the user creates/ delete directories.

○ The backup of the file system can be created by using an interface provided by the operating system.

4) Communication

In the computer system, there is a collection of processors which do not share memory peripherals devices or a clock, the operating system manages communication between all the processes. Multiple processes can communicate with every process through communication lines in the network. There are some major activities that are carried by an operating system with respect to communication.

- Two processes may require data to be transferred between the process.
- Both the processes can be on one computer or a different computer, but are connected through a computer network.

5) Error handling

An error is one part of the system that may cause malfunctioning of the complete system. The operating system constantly monitors the system for detecting errors to avoid some situations. This give relives to the user of the worry of getting an error in the various parts of the system causing malfunctioning.

The error can occur anytime and anywhere. The error may occur anywhere in the computer system like in CPU, in I/O devices or in the memory hardware. There are some activities that are performed by an operating system:

- The OS continuously checks for the possible errors.
- The OS takes an appropriate action to correct errors and consistent computing.

6) Resource management

When there are multiple users or multiple jobs running at the same time resources must be allocated to each of them. There are some major activities that are performed by an operating system:

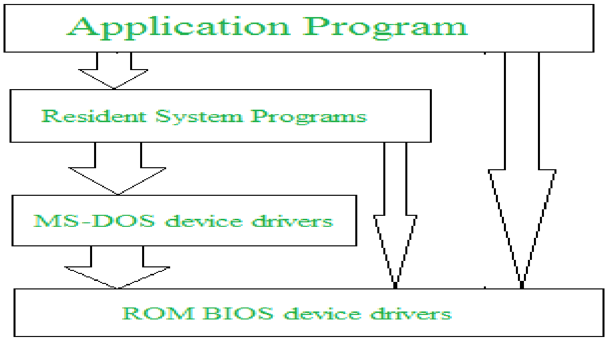
- The OS manages all kinds of resources using schedulers.
- CPU scheduling algorithm is used for better utilization of CPU.

7) Protection

The owners of information stored in a multi-user computer system want to control its use. When several disjoints processes execute concurrently it should not be possible for any process to interfere with another process. Every process in the computer system must be secured and controlled.

Operating system can be implemented with the help of various structures. The structure of the OS depends mainly on how the various common components of the operating system are interconnected and melded into the kernel. Depending on this we have following structures of the operating system: **Simple structure:**

Such operating systems do not have well defined structure and are small, simple and limited systems. The interfaces and levels of functionality are not well separated. MS-DOS is an example of such operating system. In MS-DOS application programs are able to access the basic I/O routines. These types of operating system cause the entire system to crash if one of the user programs fails. Diagram of the structure of MS-DOS is shown below.



Advantages of Simple structure:

- It delivers better application performance because of the few interfaces between the application program and the hardware.
- Easy for kernel developers to develop such an operating system.

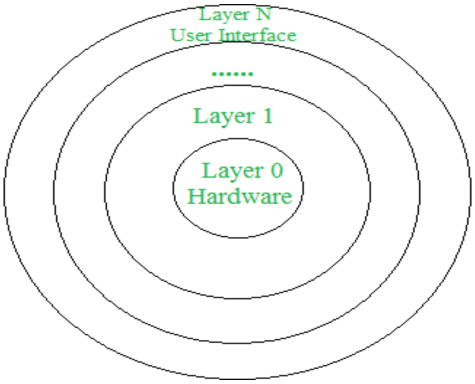
Disadvantages of Simple structure:

- The structure is very complicated as no clear boundaries exists between modules.
- It does not enforce data hiding in the operating system.

Layered structure:

An OS can be broken into pieces and retain much more control on system. In this structure the OS is broken into number of layers (levels). The bottom layer (layer 0) is the hardware and the topmost layer (layer N) is the user interface. These layers are so designed that each layer uses the functions of the lower level layers only. This simplifies the debugging process as if lower level layers are debugged and an error occurs during debugging then the error must be on that layer only as the lower level layers have already been debugged.

The main disadvantage of this structure is that at each layer, the data needs to be modified and passed on which adds overhead to the system. Moreover careful planning of the layers is necessary as a layer can use only lower level layers. UNIX is an example of this structure.



Advantages of Layered structure:

- Layering makes it easier to enhance the operating system as implementation of a layer can be changed easily without affecting the other layers.
- It is very easy to perform debugging and system verification.

Disadvantages of Layered structure:

- In this structure the application performance is degraded as compared to simple structure.
- It requires careful planning for designing the layers as higher layers use the functionalities of only the lower layers.

Introduction to Linux:

Linux is a Unix-like computer operating system assembled under the model of free and open source software development and distribution. The defining component of Linux is the Linux kernel, an operating system kernel first released 5 October 1991 by Linus Torvalds.

Linux was originally developed as a free operating system for Intel x86-based personal computers. It has since been ported to more computer hardware platforms than any other operating system. It is a leading operating system on servers and other big iron systems such as mainframe computers and supercomputers more than 90% of today's 500 fastest supercomputers run some variant of Linux, including the 10 fastest. Linux also runs on embedded systems (devices where the operating system is typically built into the firmware andhighly tailored to the system) such as mobile phones, tablet computers, network routers, televisions and video game consoles; the Android system in wide use on mobile devices is built on the Linux kernel.

Basic Features

Following are some of the important features of Linux Operating System.

- **Portable** - Portability means software's can works on different types of hardware's in same way. Linux kernel and application programs support their installation on any kind of hardware platform.
- **Open Source** - Linux source code is freely available and it is community based development project. Multiple Teams works in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
- **Multi-User** - Linux is a multiuser system means multiple users can access systemresources like memory/ ram/ application programs at same time.
- **Multiprogramming** - Linux is a multiprogramming system means multiple applicationscan run at same time.
- **Hierarchical File System** - Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** - Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, callapplication programs etc.
- **Security** - Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

Linux Advantages

1. **Low cost:** You don't need to spend time and money to obtain licenses since Linux and much of its software come with the GNU General Public License. You can start to work immediately without worrying that your software may stop working anytime because the free trial version expires. Additionally, there are large repositories from which you can freely download high quality software for almost any task you can think of.
2. **Stability:** Linux doesn't need to be rebooted periodically to maintain performance levels.It doesn't freeze up or slow down over time due to memory leaks and such. Continuous up- times of hundreds of days (up to a year or more) are not uncommon.
3. **Performance:** Linux provides persistent high performance on workstations and on networks. It can handle unusually large numbers of users simultaneously, and can make old computers sufficiently responsive to be useful again.
4. **Network friendliness:** Linux was developed by a group of programmers over the Internet and has therefore strong support for network functionality; client and server systems can be easily set up on any computer running Linux. It can perform tasks such as network backups faster and

more reliably than alternative systems.

5. **Flexibility:** Linux can be used for high performance server applications, desktop applications, and embedded systems. You can save disk space by only installing the components needed for a particular use. You can restrict the use of specific computers by installing for example only selected office applications instead of the whole suite.

6. **Compatibility:** It runs all common UNIX software packages and can process all common file formats.

7. **Choice:** The large number of Linux distributions gives you a choice. Each distribution is developed and supported by a different organization. You can pick the one you like best; the core functionalities are the same; most software runs on most distributions.

8. **Fast and easy installation:** Most Linux distributions come with user-friendly installation and setup programs. Popular Linux distributions come with tools that make installation of additional software very user friendly as well.

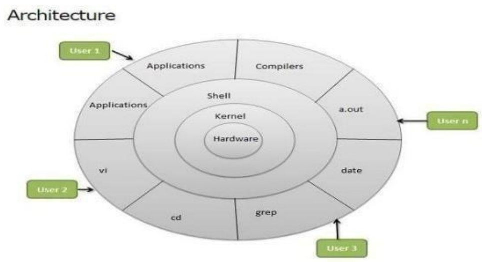
9. **Full use of hard disk:** Linux continues work well even when the hard disk is almost full.

Multi-tasking: Linux is designed to do many things at the same time; e.g., a large printing job in the background won‘t slow down your other work.

10. **Security:** Linux is one of the most secure operating systems. —Walls‖ and flexible file access permission systems prevent access by unwanted visitors or viruses. Linux users have to option to select and safely download software, free of charge, from online repositories containing thousands of high quality packages. No purchase transactions requiring credit card numbers or other sensitive personal information are necessary.

11. **Open Source:** If you develop software that requires knowledge or modification of the operating system code, LINUX‘s source code is at your fingertips. Most Linux applications are Open Source as well.

Layered Architecture:



Linux System Architecture is consists of following layers

- **Hardware layer -** Hardware consists ofall peripheral devices (RAM/ HDD/ CPU etc).

- **Kernel** - Core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
- **Shell** - An interface to kernel, hiding complexity of kernel's functions from users. Takescommands from user and executes kernel's functions.
- **Utilities** - Utility programs giving user most of the functionalities of an operating systems.

LINUX File system

Linux file structure files are grouped according to purpose. Ex: commands, data files, documentation. Parts of a Unix directory tree are listed below. All directories are grouped under the root entry "/". That part of the directory tree is left out ofthe below diagram.

1. / – Root
   - Everysingle file and directory starts from the root directory.
   - Onlyroot user has write privilege under this directory.
   - Please note that /root is root user‘s home directory, which is not same as /.
2. /bin – User Binaries
   - Contains binary executables.
   - Common linux commands you need to use in single-user modes are located under thisdirectory.
   - Commands used by all the users ofthe system are located here.
   - For example: ps, ls, ping, grep, cp.
3. /sbin – System Binaries
   - Just like /bin, /sbin also contains binary executables.
   - But, the linux commands located under this directory are used typically by systemaministrator, for system maintenance purpose.
   - For example: iptables, reboot, fdisk, ifconfig, swapon
4. /etc – Configuration Files
   - Contains configuration files required by all programs.
   - This also contains startup and shutdown shell scripts used to start/stop individualprograms.
   - For example: /etc/resolv.conf, /etc/logrotate.conf
5. /dev – Device Files
   - Contains device files.
   - These include terminal devices, usb, or anydevice attached to the system.
   - For example: /dev/tty1, /dev/usbmon0

6. /proc – Process Information

- Contains information about system process.
- This is a pseudo filesystem contains information about running process. For example: /proc/{pid} directory contains information about the process with that particular pid.
- This is a virtual filesystem with text information about system resources. For example: /proc/uptime

7. /var – Variable Files

- var stands for variable files.
- Content ofthe files that are expected to grow can be found under this directory.
- This includes — system log files (/var/log); packages and database files (/var/lib); emails (/var/mail); print queues (/var/spool); lock files (/var/lock); temp files needed across reboots (/var/tmp);

8. /tmp – Temporary Files

- Directorythat contains temporary files created bysystem and users.
- Files under this directory are deleted when system is rebooted.

9. /usr – User Programs

- Contains binaries, libraries, documentation, and source-code for second level programs.
- /usr/bin contains binary files for user programs. If you can‘t find a user binary under /bin, look under /usr/bin. For example: at, awk, cc, less, scp
- /usr/sbin contains binary files for system administrators. If you can‘t find a system binary under /sbin, look under /usr/sbin. For example: atd, cron, sshd, useradd, userdel
- /usr/lib contains libraries for /usr/bin and /usr/sbin
- /usr/local contains users programs that you install from source. For example, when youinstall apache from source, it goes under /usr/local/apache2

10. /home – Home Directories

- Home directories for allusers to store their personal files.
- For example: /home/john, /home/nikita

11. /boot – Boot Loader Files

- Contains boot loader related files.
- Kernel initrd, vmlinux, grub files are located under /boot
- For example: initrd.img-2.6.32-24-generic, vmlinuz-2.6.32-24-generic

12. /lib – System Libraries

- Contains library files that supports the binaries located under /bin and /sbin
- Library filenames are either ld* or lib*.so.*

- For example: ld-2.11.1.so, libncurses.so.5.7

13. /opt – Optional add-on Applications

- opt stands for optional.
- Contains add-on applications from individual vendors.
- add-onapplications should be installed under either /opt/ or /opt/ sub-directory.

14. /mnt – Mount Directory

- Temporary mount directorywhere sysadmins can mount filesystems.

15. /media – Removable Media Devices

- Temporary mount directory for removable devices.
- For examples, /media/cdrom for CD-ROM; /media/floppy for floppy drives; /media/cdrecorder for CD writer

16. /srv – Service Data

- srv stands for service.
- Contains server specific services related data.
- For example, /srv/cvs contains CVS related data.

File Handling Utilities

In Linux, most of the operations are performed on files. And to handle these files Linux has directories also known as folders which are maintained in a tree-like structure. Though, these directories are also a type of file themselves. Linux has 3 types of files:

1. **Regular Files:** It is the common file type in Linux. it includes files like – text files, images, binary files, etc. Such files can be created using the touch command. They consist of the majority of files in the Linux/UNIX system. The regular file contains ASCII or Human Readable text, executable program binaries, program data and much more.
2. **Directories:** Windows call these directories as folders. These are the files that store the list of file names and the related information. The root directory(/) is the base of the system, /home/ is the default location for user's home directories, /bin for Essential User Binaries, /boot – Static Boot Files, etc. We could create new directories with mkdir command.
3. **Special Files:** Represents a real physical device such as a printer which is used for IO operations. Device or special files are used for device Input/Output(I/O) on UNIX and Linux systems. You can see them in a file system like an ordinary directory or file.

In Unix systems, there are two types of special files for each device, i.e. character special files and block special files. For more details, read the article **Unix file system**.

1. Files Listing

To perform Files listings or to list files and directories ls command is used

```
$ls
```

```
$ls -l
```

2. Creating Files

touch command can be used to create a new file. It will create and open a new blank file if the file with a filename does not exist. And in case the file already exists then the file will not be affected.

$touch filename

3. Displaying File Contents

cat command can be used to display the contents of a file. This command will display the contents of the 'filename' file. And if the output is very large then we could use more or less to fit the output on the terminal screen otherwise the content of the whole file is displayed at once.

```
$cat filename
```

4. Copying a File

cp command could be used to create the copy of a file. It will create the new file in destination with the same name and content as that of the file 'filename'.

```
$cp source/filename destination/
```

5. Moving a File

mv command could be used to move a file from source to destination. It will remove the file filename from the source folder and would be creating a file with the same name and content in the destination folder.

```
$mv source/filename destination/
```

6. Renaming a File

mv command could be used to rename a file. It will rename the filename to new_filename or in other words, it will remove the filename file and would be creating a new file with the new_filename with the same content and name as that of the filename file.

```
$mv filename new_filename
```

7. Deleting a File

rmcommand could be used to delete a file. It will remove the filename file from the directory.

```
$rm filename
```

Process Utilities

A program/command when executed, a special instance is provided by the system to the process. This instance consists of all the services/resources that may be utilized by the process under execution.

- Whenever a command is issued in Unix/Linux, it creates/starts a new process. For example, pwd when issued which is used to list the current directory location the user is in, a process starts.
- Through a 5 digit ID number Unix/Linux keeps an account of the processes, this number is call process ID or PID. Each process in the system has a unique PID.
- Used up pid's can be used in again for a newer process since all the possible combinations are used.
- At any point of time, no two processes with the same pid exist in the system because it is the pid that Unixusestotrackeachprocess.

Initializing a process

A process can be run in two ways:

Method 1: Foreground Process : Every process when started runs in foreground by default, receives input from the keyboard, and sends output to the screen. When issuing pwd command

$ ls pwd

Output:

```
$ /home/geeksforgeeks/root
```

When a command/process is running in the foreground and is taking a lot of time, no other processes can be run or started because the prompt would not be available until the program finishes processing and comes out.

Method 2: Background Process: It runs in the background without keyboard input and waits till keyboard input is required. Thus, other processes can be done in parallel with the process running in the background since they do not have to wait for the previous process to be completed. Adding & along with the command starts it as a background process

$ pwd &

Since pwd does not want any input from the keyboard, it goes to the stop state until moved to the foreground and given any data input. Thus, on pressing Enter:

Output:

```
[1]   +   Done                  pwd
    $
```

That first line contains information about the background process – the job number and the process ID. It tells you that the ls command background process finishes successfully. The second is a prompt for another command.

Tracking ongoing processes

ps (Process status) can be used to see/list all the running processes.

$ ps

```
PID       TTY       TIME        CMD
19        pts/1     00:00:00    sh
24        pts/1     00:00:00    ps
```

For more information -f (full) can be used along with ps

$ ps –f

```
UID       PID  PPID C STIME     TTY        TIME CMD
52471      19      1 0 07:20     pts/1  00:00:00f      Sh
52471      25     19 0 08:04     pts/1  00:00:00       ps -f
```

For single-process information, ps along with process id is used

$ ps 19

```
PID       TTY       TIME        CMD
19        pts/1     00:00:00    sh
```

For a running program (named process) **Pidof** finds the process id's (pids)

Fields described by ps are described as:

- **UID**: User ID that this process belongs to (the person running it)
- **PID**: Process ID
- **PPID**: Parent process ID (the ID of the process that started it)
- **C**: CPU utilization of process
- **STIME**: Process start time
- **TTY**: Terminal type associated with the process
- **TIME**: CPU time is taken by the process
- **CMD**: The command that started this process

There are other options which can be used along with ps command :

- **-a**: Shows information about all users

- **-x**: Shows information about processes without terminals
- **-u**: Shows additional information like -f option
- **-e**: Displays extended information

Stoppingaprocess:

When running in foreground, hitting Ctrl + c (interrupt character) will exit the command. For processes running in background kill command can be used if it's pid is known.

Networking Commands

One can use a variety of network tools to perform tasks such as obtaining information about other systems on your network, accessing other systems, and communicating directly with other users. Network information can be obtained using utilities such as **ping, finger, traceroute, host, dig, nslookup etc**. These are useful for smaller networks and enables to access remote systems directly to copy files or execute the command.

Network Information Tools are listed below:

1. ping: The ping command is used to check if a remote system is running or up. In short this command is used to detect whether a system is connected to the network or not. **Syntax:**

```
$ ping www.geeksforgeeks.com
```

Note: In place of using domain name you can use IP address also. A *ping* operation can fail if *ping* access is denied by a network firewall.

2. host: This command is used to obtain network address information about a remote system connected to your network. This information usually consists of system's IP address, domain name address and sometimes mail server also. **Syntax:**

```
$ host www.google.com
```

3. finger: One can obtain information about the user on its network and the **who** command to see what users are currently online on your system. The who command list all users currently connected, along with when, how long, and where they logged in. **finger** can operate on large networks, though most systems block it for security reasons.

Syntax:

```
$ finger www.ABC.com
```

In place of ABC you can use any website domain or IP address.

4. traceroute: This command is use to track the sequence of computer networks. You can track to check the route through which you are connected to a host. **mtr** or **xmtr** tools can also be used to perform both *ping* and *traces*. Options are available for specifying parameters like the type of service (**-t**) or the source host (**-s**).

5. netstat: This command is used to check the status of ports whether they are open, closed, waiting and masquerade connections. Network Statistic (netstat) command display connection infor mation, routing table information etc.

Syntax:

```
$ netstat
```

Note: To display routing table information use (**netstat -r**).

6. tracepath: tracepath performs a very similar function to that of traceroute command. The main difference between these command is that tracepath doesn't take complicated options. This command doesn't require root privileges.

Syntax:

```
$ tracepath www.google.com
```

7. dig: dig(Domain Information Groper) query DNS related information like a record, cname, mxrecord etc. This command is used to solve DNS related queries.
Syntax:
$ dig www.google.com

8. hostname: This command is used to see the hostname of your computer. You can change hostname permanently in etc/sysconfig/network. After changing the hostname you need to reboot the computer. **Syntax:**
$ hostname

9. route: The route command is used to display or modify the routing table. To add a gateway use (**-n**). **Syntax:**
$ route -n

10. nslookup: You can use nslookup(name server lookup) command to find out DNS related query or testing and troubleshooting DNS server.
Syntax:
$ nslookup google.com

Filters in Linux

Filters are programs that take plain text(either stored in a file or produced by another program) as standard input, transforms it into a meaningful format, and then returns it as standard output. Linux has a number of filters. Some of the most commonly used filters are explained below:

1. cat : Displays the text ofthe file line by line.
Syntax:
cat [path]

2. head : Displays the first n lines of the specified text files. If the number of lines is not specified then by default prints first 10 lines.
Syntax:
head [-number_of_lines_to_print] [path]

3. tail : It works the same way as head, just in reverse order. The only difference in tail is, it returns the lines from bottom to up.
Syntax:
tail [-number_of_lines_to_print] [path]

4. sort : Sorts the lines alphabetically by default but there are many options available to modify the sorting mechanism. Be sure to check out the main page to see everything it can do.
Syntax:
sort [-options] [path]

5. uniq : Removes duplicate lines. uniq has a limitation that it can only remove continuous duplicate lines(although this can be fixed by the use of piping). Assuming we have the following data.

Syntax:
uniq [options] [path]

When applying uniq to sorted data, it removes the duplicate lines because, after sorting data, duplicate lines come together.

6. wc : wc command gives the number of lines, words and characters in the data.
Syntax:
wc [-options] [path]

In above image the wc gives 4 outputs as:

- number of lines
- number of words
- number of characters
- path

7. grep : grep is used to search a particular information from a text file.

Syntax:
grep [options] pattern [path]

Below are the two ways in which we can implement grep.

8. tac : tac is just the reverse of cat and it works the same way, i.e., instead of printing from lines 1 through n, it prints lines n through 1. It is just reverse of cat command.

Syntax:
tac [path]

9. sed : sed stands for stream editor. It allows us to apply search and replace operation on our data effectively. *sed* is quite an advanced filter and all its options can be seen on its man page.

Syntax:
sed [path]

10. nl : nl is used to number the lines of our text data.

Syntax:
nl [-options] [path]

UNIT – II

Linux: Introduction to shell, Types of shell's , example shell programs.

Process and CPU Scheduling - Process concepts and scheduling, Operations on processes, Cooperating Processes, Threads, Scheduling Criteria, Scheduling Algorithms, Multiple -Processor Scheduling.

Shell Programming

The shell has similarities to the DOS command processor Command.com (actually Dos was design as a poor copy of UNIX shell), it's actually much more powerful, really a programming language in its own right.

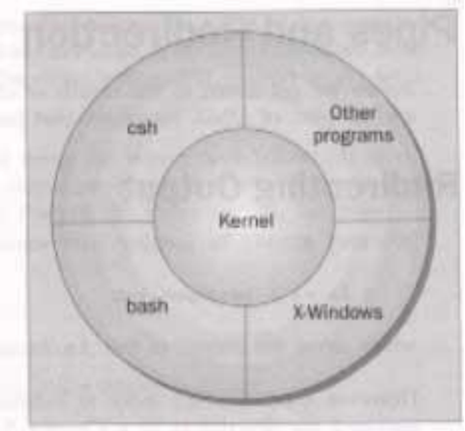
A shell is always available on even the most basic UNIX installation. You have to go through the shell to get other programs to run. You can write programs using the shell. You use the shell to administrate you r UNIX system. For example:

```
ls -al | more
```

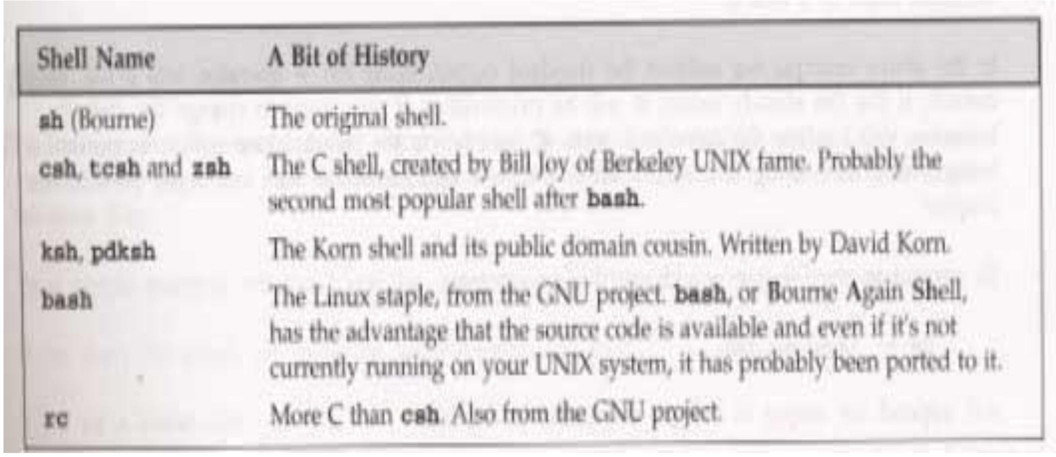
is a short shell program to get a long listing of the present directory and route the output through the more command.

What is a Shell?

A **shell** is a program that acts as the interface between you and the UNIX system, allowing you to enter commands for the operating system to execute.



Here are some common shells.



Shell Name	A Bit of History
sh (Bourne)	The original shell.
csh, **tcsh** and **zsh**	The C shell, created by Bill Joy of Berkeley UNIX fame. Probably the second most popular shell after **bash**.
ksh, **pdksh**	The Korn shell and its public domain cousin. Written by David Korn.
bash	The Linux staple, from the GNU project. **bash**, or Bourne Again Shell, has the advantage that the source code is available and even if it's not currently running on your UNIX system, it has probably been ported to it.
rc	More C than **csh**. Also from the GNU project.

Introduction- Working with Bourne Shell

- The Bourne shell, or sh, was the default Unix shell of Unix Version 7. It was developed by Stephen Bourne, of AT&T Bell Laboratories.
- A Unix shell, also called "the command line", provides the traditional user interface forthe Unix operating system and for Unix-like systems. Users direct the operation of the computer by entering command input as text for a shell to execute.
- There are many different shells in use. Theyare

– Bourne shell (sh)

– C shell (csh)

– Korn shell (ksh)Bourne Again shell (bash)

- When we issue a command the shell is the first agency to acquire the information. Itaccepts and interprets user requests. The shell examines &rebuilds the commands &leaves the execution work to kernel. The kernel handles the h/w on behalf ofthesecommands &all processes in the system.
- The shell is generally sleeping. It wakes up when an input is keyed in at the prompt. Thisinput is actually input to the program that represents the shell.

Shell responsibilities

1. Program Execution
2. Variable and Filename Substitution
3. I/O Redirection
4. Pipeline Hookup
5. Environment Control
6. Interpreted Programming Language1.Program Execution:

- The shell is responsible for the execution ofall programs that you request fromyourterminal.
- Each time you type in a line to the shell, the shell analyzes the line and thendetermineswhat to do.

- The line that is typed to the shell is known more formallyas the command line. The shell scans this command line and determines the name of the program to be executed and what arguments to pass to the program.

2. Variable and Filename Substitution:

- Like any other programming language, the shell lets you assign values to variables. Whenever you specify one of these variables on the command line, preceded by adollar sign, the shell substitutes the value assigned to the variable at that point.

3. I/O Redirection:

- It is the shell's responsibility to take care of input and output redirection on the command line. It scans the command line for the occurrence of the special redirection characters <,
>, or >>.

4. Pipeline Hookup:

- Just as the shell scans the command line looking for redirection characters, it also looks for the pipe character |. For each such character that it finds, it connects the standard output from the command preceding the | to the standard input of the one following the|. It then initiates execution of bothprograms.

5. Environment Control:

- The shell provides certain commands that let you customize your environment. Your environment includes home directory, the characters that the shell displays to prompt you to type in a command, and a list of the directories to be searched whenever you request that a program be executed.

6. Interpreted Programming Language:

- The shell has its own built-in programming language. This language is interpreted, meaning that the shell analyzes each statement in the language one line at a time and thenexecutes it. This differs from programming languages such as C and FORTRAN, in which the programming statements are typically compiled into a machine-executable form before they are executed.
- Programs developed in interpreted programming languages are typically easier to debug and modify than compiled ones. However, theyusually take much longer to execute than their compiled equivalents.

Pipes connect processes together. The input and output of UNIX programs can be redirected.

Redirecting Output

The > operator is used to redirect output ofa program. For example:ls -l> lsoutput.txt
redirects the output ofthe list command from the screen to the file lsoutput.txt.

To 0append to a file, use the >> operator.

Pipes and Redirection

```
ps >> lsoutput.txt
```

Redirecting Input

You redirect input byusing the < operator. For example:more < killout.txt

Pipes

We can connect processes together using the pipe operator (|). For example, the following program means run the ps program, sort its output, and save it in the file pssort.out

ps | sort > pssort.out
The sort command will sort the list of words in a textfile into alphbetical order according to theASCII code set character order.

Here Documents

A here document is a special way of passing input to a command from a shell script. Thedocument starts and ends with the same leader after <<. For example:

#!/bin/sh

cat < this is a heredocument
!FUNKY!

How It Works

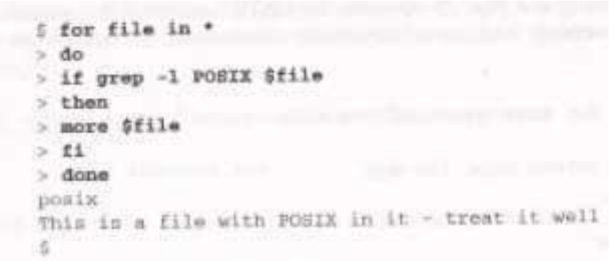
It executes the here document as if it were input commands.

Running a Shell Script

You can type in a sequence of commands and allow the shell to execute them interactively, oryouu can sotre these commands in a file which you can invoke as a program.

Interactive Programs

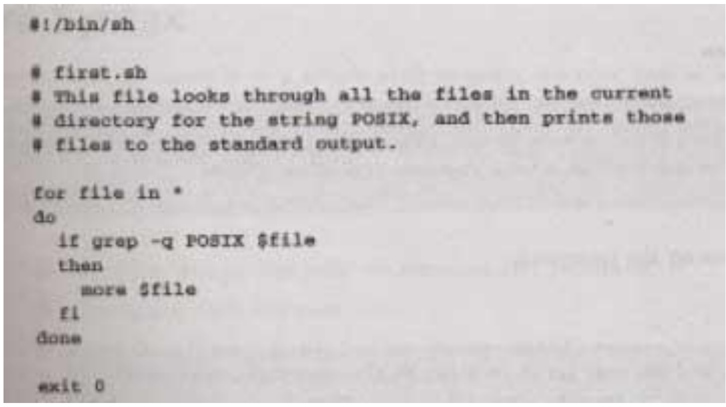
A quick way of trying out small code fragments is to just type in the shell script on the commandline. Here is a shell program to compile only files that contain the string POSIX.



```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
posix
This is a file with POSIX in it - treat it well
$
```

The hell as a Programming LanguageCreating a Script

To create a **shell script** first use a text editor to create a file containing the commands. Forexample, type the following commands and save them as first.sh



```
#!/bin/sh

# first.sh
# This file looks through all the files in the current
# directory for the string POSIX, and then prints those
# files to the standard output.

for file in *
do
  if grep -q POSIX $file
  then
    more $file
  fi
done

exit 0
```

Note: commands start with a #.

The line

#!/bin/sh

is special and tells the systemto use the /bin/sh program to execute this program.

The command
exit 0
Causes the script program to exit and return a value of 0, which means there were not errors.

Making a Script Executable

There are two ways to execute the script. 1) invoke the shell with the name of the script file as a parameter, thus:/bin/sh first.sh
Or 2) change the mode ofthe script to executable and then after execute it by just typing itsname.
chmod +x first.shfirst.sh
Actually, you may need to type:
./first.sh to make the file execute unles the path variable has your directory in it.

Shell Syntax

The modern UNIX shell can be used to write quite large, structured programs.

Shell metacharacters

The shell consists of large no. of metacharacters. These characters plays vital role in Unixprogramming.

<u>Types of metacharacters:</u>

1. File substitution

2. I/O redirection

3. Process execution

4. Quoting metacharacters

5. Positionalparameters

6. Special characters

7. Command substitution

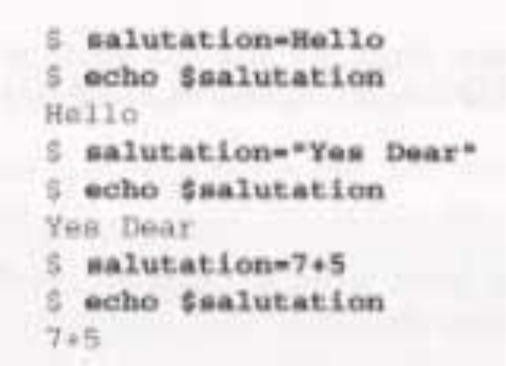
Filename substitution:

These metacharacters are used to match the filenames in a directory.

Metacharacter	significance
*	matches any no. ofcharacters
?	matches a single character
[ijk]	matches a single character either i,j,k
[!ijk]	matches a single character that is not an I,j,k

Shell Variables

Variables are generally created when you first use them. By default, all variables are considered and stored as strings. Variable names are case sensitive.



```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

U can define & use variables both in the command line and shell scripts. These variablesare called shell variables.

No type declaration is necessary before u can use a shellvariable.

Variables provide the ability to store and manipulate the information with in the shellprogram. The variables are completely under the control ofuser.

Variables in Unix are oftwo types.

1) User-defined variables:

Generalized form:

variable=value.Eg: $x=10

$echo $x10

To remove a variable use unset.

$unset x

All shell variables are initialized to null strings by default. To explicitly set null valuesuse

x= or x=‗‘ or x=―‖

To assign multiword strings to a variable use

$msg=‗u have a mail‘

2) Environment Variables

- They are initialized when the shell script starts and normallycapitalized to distinguish them fromuser-defined variables in scripts
- To display all variables in the local shell and their values, type the **set** command
- The **unset** command removes the variable from the current shell and subshell

Environment Variables	**Description**
$HOME	**Home directory**
$PATH	**List of directories to search for commands**
$PS1	**Command prompt**
$PS2	**Secondary prompt**
$SHELL	**Current login shell**
$0	**Name of the shell script**
$#	**No . of parameters passed**
$$	**Process ID of the shell script**

Command substitution and Shell commands:

r ead:

The read statement is a tool for taking input from the user i.e. making scripts interactive It is used with one or more variables. Input supplied through the standard input is read into these variables.

$read name

What ever u entered is stored in the variablename. printf:

Printf is used to print formattedo/p. printf "format" arg1 arg2 ...Eg:

$ printf "This is a number: %d\n" 10This is a number: 10

$

Printf supports conversion specification characters like %d, %s ,%x

,%o…. Exit status ofa command:

- Every command returns a value after execution .This value is called the exitstatus or return value of a command.
- This value is said to be true if the command executes successfully and false if it fails.
- There is special parameter used by the shell it is the $?. It stores the exit status of a command.

exit:

- The exit statement is used to prematurely terminate a program. When this statement is encountered in a script, execution is halted and control is returned to the calling program- in most cases the shell.
- U don‘t need to place exit at the end of every shell script because the shell knows when script execution is complete.
- Set :

Set is used to produce the list ofcurrently defined variables.

$set

□ Set is used to assign values to the positional parameters.

```
$set welcome to Unix
```

The do-nothing(:)Command

□ It is a null command.

□ In some older shell scripts, colon was used at the start of a line to introduce acomment, but modern scripts uses # now.

□ expr:

□ The expr command evaluates its arguments as an expression:

```
$ expr 8 + 6
$ x=`expr 12 / 4 `
$ echo $x3
```

export:

There is a way to make the value of a variable known to a sub shell, and that'sby exporting it with the export command. The format of this command is

export variables

where variables is the list of variable names that you want exported. For any sub shells that get executed from that point on, the value of the exported variables will bepassed down to the sub shell.

eval:

eval scans the command line twice before executing it. General form for evalis eval command-line

Eg:

```
$ cat last

eval echo \$$#

$ last one two three fourfour
```

${n}

If u supply more than nine arguments to a program, u cannot access the tenth and greater arguments with $10, $11, and so on.

${n} must be used. So to directlyaccess argument 10, you must write

${10}

Shift command:

The shift command allows u to effectively left shift your positional parameters. If u executethe command

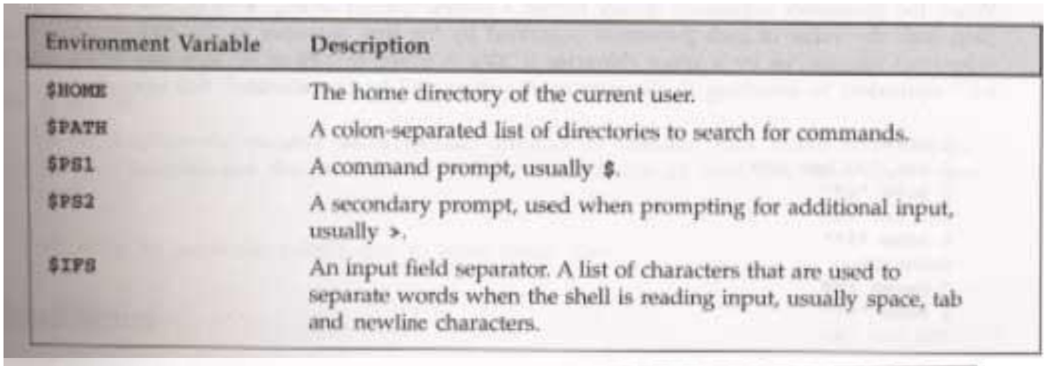
Shift

whatever was previously stored inside $2 will be assigned to $1, whatever was previouslystored in $3 will be assigned to $2, and so on. The old value of$1 will be irretrievably lost.

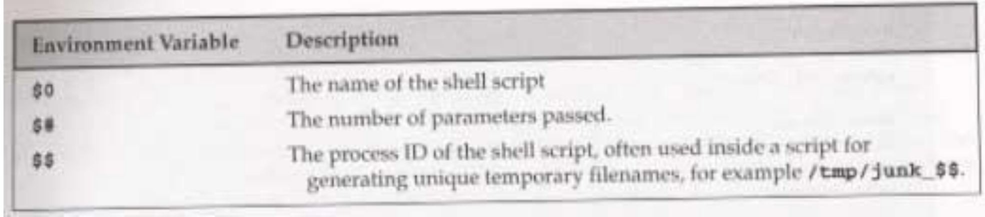
The Environment-Environment Variables

It creates the variable salutation, displays its value, and some parameter variables.

- When a shell starts, some variables are initialized from values in the environment. Here is a sample of some ofthem.



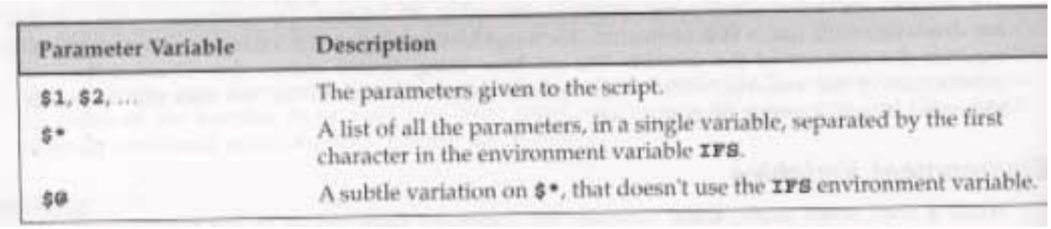
Environment Variable	Description
$HOME	The home directory of the current user.
$PATH	A colon-separated list of directories to search for commands.
$PS1	A command prompt, usually $.
$PS2	A secondary prompt, used when prompting for additional input, usually >.
$IFS	An input field separator. A list of characters that are used to separate words when the shell is reading input, usually space, tab and newline characters.



Environment Variable	Description
$0	The name of the shell script
$#	The number of parameters passed.
$$	The process ID of the shell script, often used inside a script for generating unique temporary filenames, for example /tmp/junk_$$.

Parameter Variables

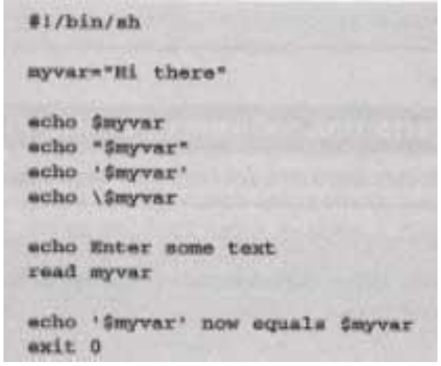
- If your script is invoked with parameters, some additional variables are created.



Parameter Variable	Description
$1, $2, ...	The parameters given to the script.
$*	A list of all the parameters, in a single variable, separated by the first character in the environment variable IFS.
$@	A subtle variation on $*, that doesn't use the IFS environment variable.

Quoting

Normally, parameters are separated by white space, such as a space. Single quot marks can be used to enclose values containing space(s). Type the following into a file called quot.sh

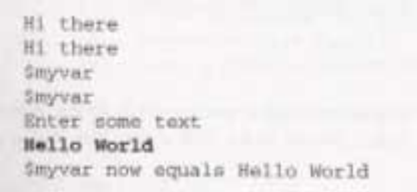


```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

make sure to make it executable bytyping the command:

< chmod a+xquot.sh The results of executing

the file is:



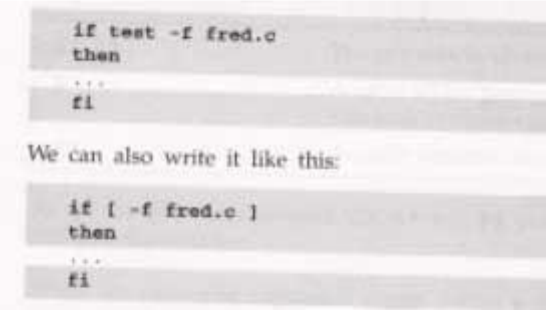
```
Hi there
Hi there
$myvar
$myvar
Enter some text
Hello World
$myvar now equals Hello World
```

How It Works

The variable myvar is created and assigned the string Hi there. The content of the variable is displyed using the echo $. Double quotes don't effect echoing the value. Single quotes and backslash do.

The test, or []Command

Here is how to check for the existance ofthe file fred.c using the test and using the []command.

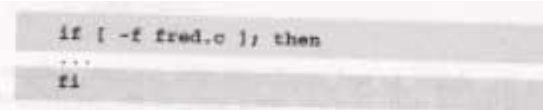


```
if test -f fred.c
then
...
fi
```

We can also write it like this:

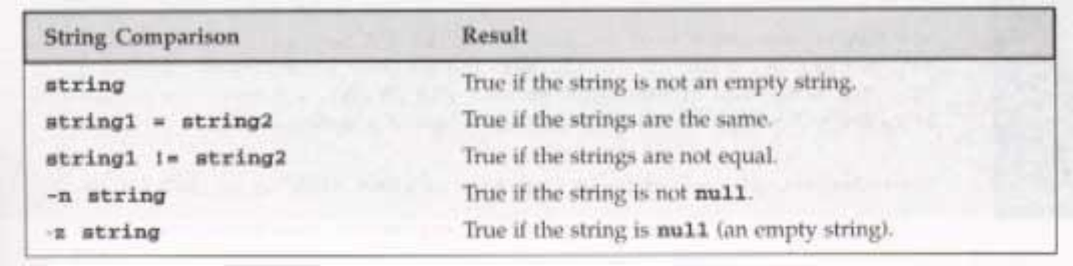
```
if [ -f fred.c ]
then
...
fi
```

You can even place the then on the same line as the if, if youu add a semicolon before theword then.



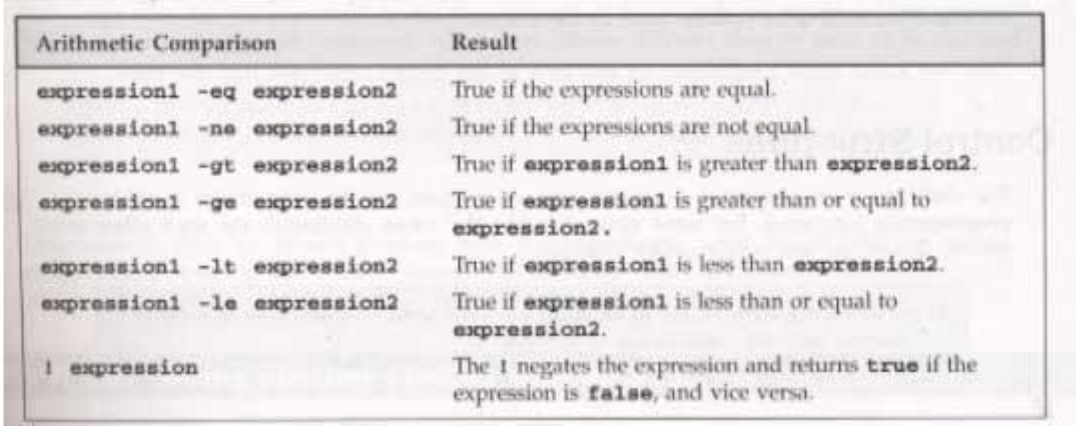
```
if [ -f fred.c ]; then
...
fi
```

Here are the conditon types that can be used with the test command. There are stringcomparison.



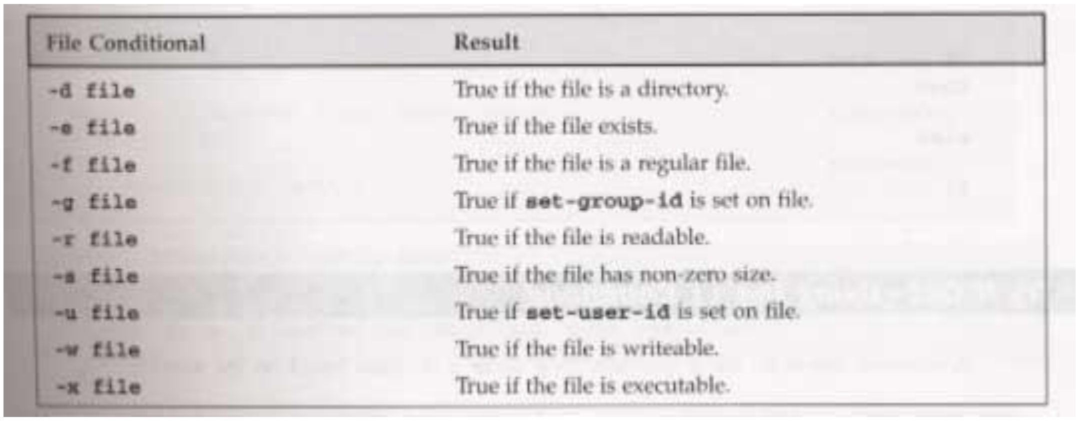
String Comparison	Result
`string`	True if the string is not an empty string.
`string1 = string2`	True if the strings are the same.
`string1 != string2`	True if the strings are not equal.
`-n string`	True if the string is not `null`.
`-z string`	True if the string is `null` (an empty string).

There are arithmetic comparison.



Arithmetic Comparison	Result
`expression1 -eq expression2`	True if the expressions are equal.
`expression1 -ne expression2`	True if the expressions are not equal.
`expression1 -gt expression2`	True if `expression1` is greater than `expression2`.
`expression1 -ge expression2`	True if `expression1` is greater than or equal to `expression2`.
`expression1 -lt expression2`	True if `expression1` is less than `expression2`.
`expression1 -le expression2`	True if `expression1` is less than or equal to `expression2`.
`! expression`	The `!` negates the expression and returns `true` if the expression is `false`, and vice versa.

There are file conditions.



File Conditional	Result
`-d file`	True if the file is a directory.
`-e file`	True if the file exists.
`-f file`	True if the file is a regular file.
`-g file`	True if `set-group-id` is set on file.
`-r file`	True if the file is readable.
`-s file`	True if the file has non-zero size.
`-u file`	True if `set-user-id` is set on file.
`-w file`	True if the file is writeable.
`-x file`	True if the file is executable.

Control Structures

The shell has a set of control structures.

if

The if statement is vary similar other programming languages except it ends with a fi.if condition then

```
else fi

statementsstatements
```

elif

the elif is better known as "else if". It replaces the else part of an if statement with another if statement. You can try it out by using the following script.

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"read timeofday

if [ $ti0meofday = "yes" ]then
echo "Good morning"elif [ $timeofday = "no" ]; thenecho "Good afternoon"
else
echo "Sorry, $timeofday not recognized. Enter yesor no" exit 1 fi

exit 0
```

How It Works

The above does a second test on the variable timeofday if it isn't equalto yes.

A Problem with Variables

If a variable is set to null, the statement
if [$timeofday = "yes"]if [= "yes"]

which is illegal. This problem can be fixed by using double quotes around the variable name. if ["$timeofday" = "yes"]
.

for

The for construct is used for looping through a range of values, which can be any set of strings. The syntax is:

```
for variable in valuesdo
statements
done
```

Tryout the following script:

```
#!/bin/sh

for foo in bar fud 43do
echo $foo
done exit 0
```

When executed, the output should be:

```
bar fud043
```

How It Works

The above example creates the variable foo and assigns it a different value each time around thefor loop.

How It Works

Here is another script which uses the $(command) syntax to expand a list to chap3.txt, chap4.txt, and chap5.txt and print the files.

```
#!/bin/sh

for file in $(ls chap[345].txt); dolpr $file
done0
```

while

While loops will loop as long as some condition exist. OF course something in the body statements of the loop should eventually change the condition and cause the loop to exit. Here is the while loop syntax.

```
while condition do
statements
done
```

Here is a whil loop that loops 20 times.#!/bin/sh

```
foo=1

while [ "$foo" -le 20 ]do
done exit 0
```

How It Works

echo "Here we go again" foo=$(($foo+1))

The above script uses the [ ] command to test foo for <= the value 20. The linefoo=$(($fo0o+1)) increments the value of foo each time the loop executes..

until

The until statement loops until a condition becomes true! Its syntax is:

```
until conditiondo
statements
done
```

Here is a script using until.

```
#!/bin/sh

untilwho | grep "$1" > /dev/nulldo
Sl0eep 60
done

# now ring the bell and announce the expected user.

echo -e \\a
echo "**** $1 has just loogged in ****"exit 0
```

case

The case statement allows the testing of a variable for more then one value. The case statementends with the word esac. Its syntax is:

```
case variable in
pattern [ | pattern] ...) statements;;pattern [ | pattern] ...) statements;;
...
esac
```

Here is a sample script using a case statement:

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"read timeofday

case "$timeofday" in
"yes") echo "Good Morning";; "no" ) echo "Good Afternoon";;0"y" ) echo "Good Morning";; "n" ) echo
"Good Afternoon";;
* ) echo "Soory, answer not recognized";;
esac exit 0
```

The value in the varaible timeofday is compared to various strings. When a match is made, the associated echo command is executed.

Here is a case where multiple strings are tested at a time, to do the some action.case "$timeofday" in

```
"yes" | "y" | "yes" | "YES" ) echo "good Morning";;"n"* | "N"* ) <echo "Good Afternoon";;
* ) < echo "Sorry, answer not recognized";;
0esac
```

How It Works

The above has sever strings tested for each possible statement.

Here is a case statement that executes multiple statements for each case.case "$timeofday" in

```
"yes" | "y" |"Yes" | "YES" )echo "Good Morning"
echo "Up bright and earlythis morning"
;;
                        [nN
                        ]*)          echo "Good Afternoon"
                                     ;;

                        *)           echo "Sorry, answer not recognized" echo "Please answer yes or
                                     noo" exit 1
                                     ;;
esac
```

How It Works

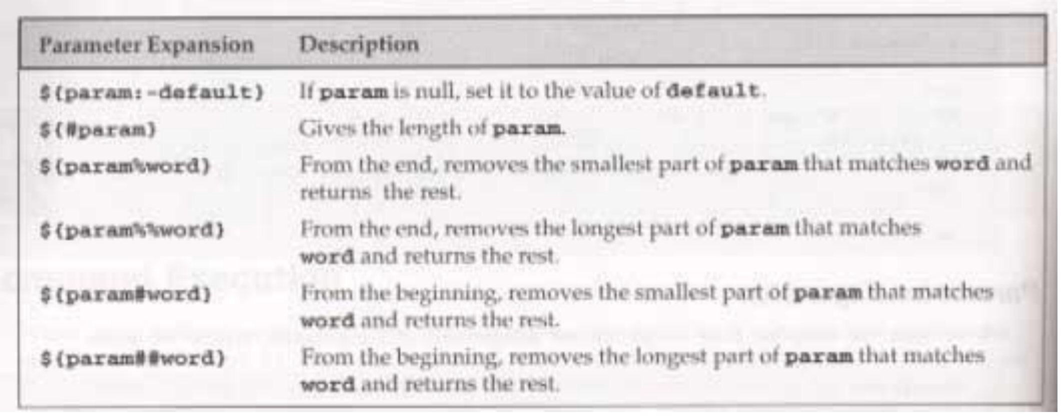
When a match is found to the variable value oftimeofday, all the statements up to the ;; areexecuted.

Parameter Expansion

Using { } around a variable to protect it against expansion.#!/bin/sh

```
for i in 1 2do
my_secret_process ${i}_tmp
done
```

Here are some ofthe parameter expansion



Parameter Expansion	Description
${param:-default}	If **param** is null, set it to the value of **default**.
${#param}	Gives the length of **param**.
${param%word}	From the end, removes the smallest part of **param** that matches **word** and returns the rest.
${param%%word}	From the end, removes the longest part of **param** that matches **word** and returns the rest.
${param#word}	From the beginning, removes the smallest part of **param** that matches **word** and returns the rest.
${param##word}	From the beginning, removes the longest part of **param** that matches **word** and returns the rest.

How It Works

The try it out exercise uses parameter expansion to demonstrate how parameter expansion works.

Shell Script ExamplesExample

```
#!/bin/sh

echo "Is it morning? (Answer yes or no)"read timeofday

if [ $timeofday = "yes" ]; then
echo "Good Morning"

else

echo "Good afternoon"

fi exit 0
```

elif - Doing further Checks

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"read timeofday

if [ $timeofday = "yes" ]; thenecho "Good Morning"
```

```
elif [ $timeofday = "no" ]; thenecho "Good afternoon"

else        echo "Wrong answer! Enter yes or no"exit 1

fi          exit 0
```

Process

A process is a program at the time of execution.

Differences between Process and Program

Process	Program
Process is a dynamic object	Program is a static object
Process is sequence of instruction execution	Program is a sequence of instructions
Process loaded in to main memory	Program loaded into secondarystorage devices
Time span ofprocess is limited	Time span of program is unlimited
Process is a active entity	Program is a passive entity

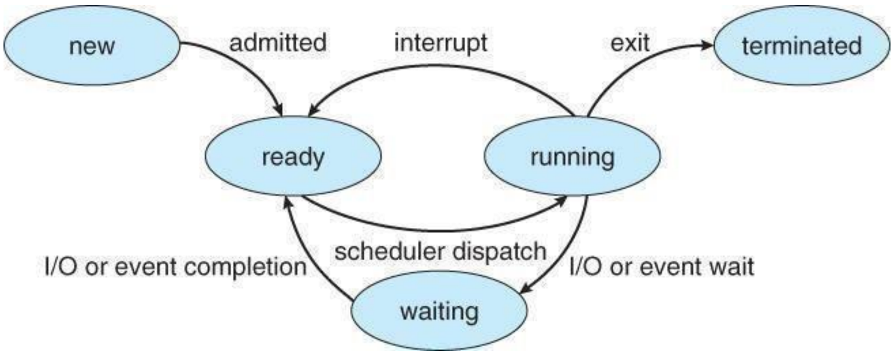
Process States

When a process executed, it changes the state, generally the state of process is determined bythe current activity of the process. Each process may be in one of the following states:

1. New : The process is being created.
2. Running : The process is being executed.
3. Waiting : The process is waiting for some event to occur.
4. Ready : The process is waiting to be assigned to a processor.
5. Terminated : The Process has finished execution.

Only one process can be running in any processor at any time, But many process may be inready and waiting states. The ready processes are loaded into a “ready queue”.

Diagram of process state



a) **New ->Ready** : OS creates process and prepares the process to be executed, then OS moved the process into ready queue.

b) **Ready->Running** : OS selects one of the Jobs from ready Queue and move themfrom ready to Running.

c) **Running->Terminated** : When the Execution of a process has Completed, OS terminates that process from running state. Sometimes OS terminates the process for some other reasons including Time exceeded, memory unavailable, access violation, protection Error, I/O failure and soon.

d) **Running->Ready** : When the time slot of the processor expired (or) If the processor received any interrupt signal, the OS shifted Running -> Ready State.

e) **Running -> Waiting** : A process is put into the waiting state, if the process need anevent occur (or) an I/O Device require.

f) **Waiting->Ready** : A process in the waiting state is moved to readystate when the event for which it has been Completed.

Process Control Block:

Each process is represented in the operating System by a Process Control Block.

It is also called Task Control Block. It contains many pieces of information associated with a specific Process.

Process State
Program Counter
CPU Registers
CPU Scheduling Information
Memory – Management Information
Accounting Information
I/O Status Information

Process Control Block

1. **Process State** : The State may be new, ready, running, and waiting, Terminated…
2. **Program Counter** : indicates the Address of the next Instruction to be executed.
3. **CPUregisters** : registers include accumulators, stack pointers,

General purpose Registers…

4. **CPU-Scheduling Info** : includes a process pointer, pointers to scheduling Queues, other scheduling parameters etc.
5. **Memory management Info**: includes page tables, segmentation tables, value ofbase and limit registers.
6. **Accounting Information:** includes amount of CPU used, time limits, Jobs(or)Process numbers.
7. **I/O Status Information**: Includes the list of I/O Devices Allocated to the processes, list ofopen files.

Threads:

A process is divide into number of light weight process, each light weight process is said to be a Thread. The Thread has a program counter (Keeps track of which instruction to execute next), registers (holds its current working variables), stack (execution History).

Thread States:

1. born State : Athread is just created.
2. ready state : The thread is waiting for CPU.
3. running : Systemassigns the processor to the thread.
4. sleep : A sleeping thread becomes readyafter the designated sleep time expires**.**
5. dead : The Execution ofthe thread finished.

Egg: Word processor.
Typing, Formatting, Spell check, saving are threads.

Differences between Process and Thread

Process	**Thread**
Process takes more time to create.	Thread takes less time to create.
it takes more time to complete execution & terminate.	Less time to terminate.
Execution is very slow.	Execution is very fast.
It takes more time to switch b/w two processes.	It takes less time to switch b/w two threads.
Communication b/w two processes is difficult .	Communication b/w two threads is easy.
Process can’t share the same memory area.	Threads can share same memory area.
System calls are requested to communicate each other.	System calls are not required.
Process is loosely coupled.	Threads are tightly coupled.
It requires more resources to execute.	Requires few resources to execute.

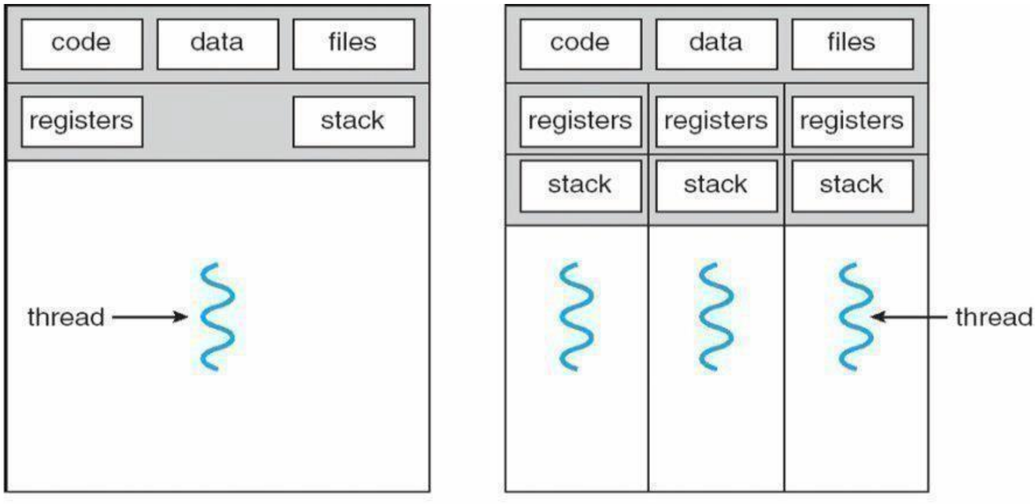
Multithreading

A process is divided into number of smaller tasks each task is called a Thread. Number ofThreads with in a Process execute at a time is called Multithreading.

If a program, is multithreaded, even when some portion of it is blocked, the whole program is not blocked. The rest of the program continues working If multiple CPU's are available.

Multithreading gives best performance. If we have only a single thread, number of CPU's available, No performance benefits achieved.

- Process creation is heavy-weight while thread creation is light-weight Can simplify code, increase
□ efficiency



single-threaded process multithreaded process

□ Kernels are generally multithreaded

CODE- Contains instruction

DATA- holds global variable

FILES-opening and closing files

REGISTER- contain information about CPU state

STACK-parameters, local variables, functions

PROCESS SCHEDULING:

CPU is always busy in **Multiprogramming**. Because CPU switches from one job to another job. But in **simple computers** CPU sit idle until the I/O request granted.

scheduling is a important OS function. All resources are scheduled before use.(cpu, memory, devices…..) Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory ata time and the loaded process shares the CPU using time multiplexing

. **Scheduling Objectives**

Maximize throughput.

Maximize number of users receiving acceptable response times.Be predictable.

Balance resource use.

Avoid indefinite postponement.

Enforce Priorities.

Give preference to processes holding key resources

SCHEDULING QUEUES: people live in rooms. Process are present in rooms knows as queues. There are 3types

1. **job queue**: when processes enter the system, they are put into a **job queue**, which consists all processes in the system. Processes in the job queue reside on mass storage and await the allocation of main memory.
2. **ready queue**: if a process is present in main memory and is ready to be allocated tocpu for execution, is kept in **ready queue.**
3. **device queue**: if a process is present in waiting state (or) waiting for an i/o event tocomplete is said to bein device queue.(or)

The processes waiting for a particular I/O device is called device queue.

Schedulers : There are 3 schedulers

1. Long termscheduler.
2. Mediumtermscheduler
3. Short term scheduler.

Scheduler duties:

- Maintains the queue.
- Select the process from queues assign to CPU.

Types of schedulers

1. **Long term scheduler:**

select the jobs from the job pooland loaded these jobs into main memory (ready queue).Long term scheduler is also called job scheduler.

2. **Short term scheduler:**

select the process from ready queue, and allocates it to the cpu.

If a process requires an I/O device, which is not present available then process enters devicequeue.

short term scheduler maintains ready queue, device queue. Also called as cpu scheduler.

3. **Medium term scheduler**: if process request an I/O device in the middle of the execution, then the process removed from the main memory and loaded into the waiting queue. When the I/O operation completed, then the job moved from waiting queue to ready queue. These two operations performed by medium term scheduler.

Context Switch: Assume, main memory contains more than one process. If cpu is executing a process, if time expires or if a high priority process enters into main memory, then the scheduler saves information about current process in the PCB and switches to execute the another process. The concept of moving CPU by scheduler from one process to other process is known as context switch.

Non-Preemptive Scheduling: **CPU** is assigned to one process, CPU do not release until the competition of that process. The CPU will assigned to some other process only after the previous process has finished.

Preemptive scheduling: here CPU can release the processes even in the middle of the execution. CPU received a signal from process p2. OS compares the priorities of p1 ,p2. If p1>p2, CPU continues the execution of p1. If p1<p2 CPU preempt p1 and assigned to p2.

Dispatcher: The main job of dispatcher is switching the cpu from one process to another process. Dispatcher connects the cpu to the process selected by the short term scheduler.

Dispatcher latency: The time it takes by the dispatcher to stop one process and start another process is known as dispatcher latency. If the dispatcher latency is increasing, then the degree of multiprogramming decreases.

SCHEDULING CRITERIA:

1. **Throughput**: how many jobs are completed bythe cpu with in a timeperiod.
2. **Turn around time** : The time interval between the submission of the processand time of the completion is turn around time.

TAT = Waiting time in ready queue + executing time + waiting time in waiting queue forI/O.

3. **Waiting time**: The time spent bythe process to wait for cpu to beallocated.
4. **Response time**: Time duration between the submission and firstresponse.
5. **Cpu Utilization**: CPU is costly device, it must be kept as busy aspossible. Eg: CPU efficiency is 90% means it is busy for 90 units, 10 units idle.

CPU SCHEDULINGALGORITHMS:

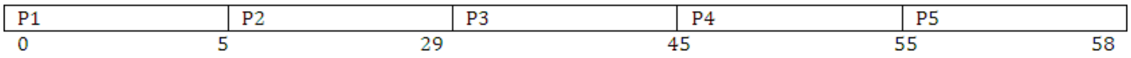
1. First come First served scheduling: (FCFS): The process that request the CPU first is holds the cpu first. If a process request the cpu then it is loaded into the ready queue, connect CPU to that process.

Consider the following set of processes that arrive at time 0, the length of the cpu burst time given in milli seconds.

burst time is the time, required the cpu to execute that job, it is in milli seconds.

Process	**Burst time(milliseconds)**
P1	5
P2	24
P3	16
P4	10
P5	3

Chart:



Average turn around time:

Turnaround time= waiting time + burst time

Turn around time for p1= 0+5=5.
Turn around time for p2=5+24=29
Turn around time for p3=29+16=45 Turn around time for p4=45+10=55 Turn around time for p5= 55+3=58
Average turn around time= (5+29++45+55+58/5) = 187/5 =37.5 millisecounds

Average waiting time:

waiting time= starting time- arrival time

Waiting time for p1=0
Waiting time for p2=5-0=5 Waiting time for p3=29-0=29 Waiting time for p4=45-0=45 Waiting time for p5=55-0=55
Average waiting time= 0+5+29+45+55/5 = 125/5 = 25 ms.

Average Response Time :

Formula : First Response - Arrival Time Response Time for P1 =0
Response Time for P2 => 5-0 = 5
Response Time for P3 => 29-0 = 29
Response Time for P4 => 45-0 = 45
Response Time for P5 => 55-0 = 55
Average Response Time => (0+5+29+45+55)/5 =>25ms

1) First Come FirstServe:

It is Non Primitive Scheduling Algorithm.

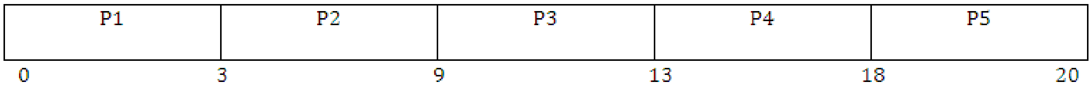
PROCESS	**BURST TIME**	**ARRIVAL TIME**
P1	3	0
P2	6	2
P3	4	4

P4	5	6
P5	2	8

Process arrived in the order P1, P2, P3, P4, P5.

P1 arrived at 0 ms.

P2 arrived at 2 ms. P3 arrived at 4 ms. P4 arrived at 6 ms. P5 arrived at 8 ms.



Average Turn Around Time

Formula : Turn around Time =**:** waiting time + burst time Turn Around Time for P1 => 0+3= 3

Turn Around Time for P2 => 1+6 = 7

Turn Around Time for P3 => 5+4 = 9

Turn Around Time for P4 => 7+ 5= 12

Turn Around Time for P5 => 2+ 10=12

Average Turn Around Time => (3+7+9+12+12)/5 =>43/5 = 8.50 ms.

Average Response Time :

Formula : Response Time = First Response - Arrival Time

Response Time of P1 = 0

Response Time of P2 => 3-2 = 1

Response Time of P3 => 9-4 = 5

Response Time of P4 => 13-6 = 7

Response Time of P5 => 18-8 =10

Average Response Time => (0+1+5+7+10)/5 => 23/5 = 4.6 ms

Advantages: Easy to Implement, Simple.

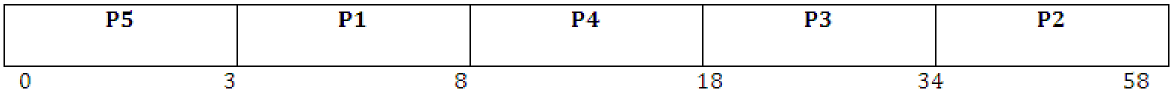
Disadvantage: Average waiting time is very high.

2) Shortest Job First Scheduling (SJF):

Which process having the smallest CPU burst time, CPU is assigned to that process . If two process having the same CPU burst time, FCFS is used.

PROCESS	CPU BURST TIME
P1	5

P2	24
P3	16
P4	10
P5	3



P5 having the least CPU burst time (3ms). CPU assigned to that (P5). After completion of P5 short term scheduler search for nest (P1).......

Average Waiting Time :

Formula = Staring Time - Arrival Time

waiting Time for P1 => 3-0 = 3

waiting Time for P2 => 34-0 = 34

waiting Time for P3 => 18-0 = 18

waiting Time for P4 =>8-0=8

waiting time for P5=0

Average waiting time => (3+34+18+8+0)/5 => 63/5 =12.6 ms

Average Turn Around Time :

Formula = waiting Time + burst Time

Turn Around Time for P1 => 3+5 =8

Turn Around for P2 => 34+24 =58

Turn Around for P3 => 18+16 = 34

Turn Around Time for P4 => 8+10 =18

Turn Around Time for P5 => 0+3 = 3

Average Turn around time => (8+58+34+18+3)/5 => 121/5 = 24.2 ms

Average Response Time :

Formula : First Response - Arrival Time

First Response time for P1 =>3-0 = 3

First Response time for P2 => 34-0 = 34

First Response time for P3 => 18-0 = 18

First Response time for P4 => 8-0 = 8

First Response time for P5 = 0

Average Response Time => (3+34+18+8+0)/5 => 63/5 = 12.6 ms

SJF is Non primitive scheduling algorithm

Advantages : **Least average waiting time**

Least average turn around time Least average response time

Average waiting time (FCFS) = 25 ms

Average waiting time (SJF) = 12.6 ms 50% time saved in SJF.

Disadvantages:

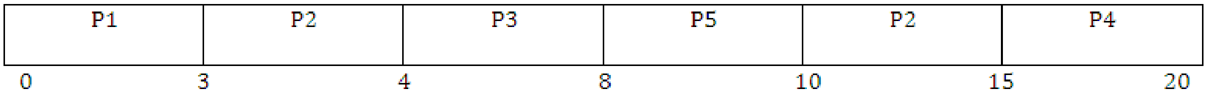
- Knowing the length ofthe next CPU burst time is difficult.
- Aging (Big Jobs are waiting for long time for CPU)

3) Shortest Remaining Time First (SRTF):

This is primitive scheduling algorithm.

Short term scheduler always chooses the process that has term shortest remaining time. When a new process joins the ready queue , short term scheduler compare the remaining time of executing process and new process. If the new process has the least CPU burst time, The scheduler selects that job and connect to CPU. Otherwise continue the old process.

PROCESS	BURST TIME	ARRIVAL TIME
P1	3	0
P2	6	2
P3	4	4
P4	5	6
P5	2	8



P1 arrives at time 0, P1 executing First , P2 arrives at time 2. Compare P1 remaining time and P2 (3 -2 = 1) and 6. So, continue P1 after P1, executing P2, at time 4, P3 arrives, compare P2 remaining time (6-1=5) and 4 (4<5) .So, executing P3 at time 6, P4 arrives. Compare P3 remaining time and P4 (4 - 2=2) and 5 (2<5). So, continue P3 , after P3, ready queue consisting P5 is the least out of three. So execute P5, next P2, P4.

FORMULA : Finish time - Arrival

Time Finish Time for P1 => 3-0 = 3

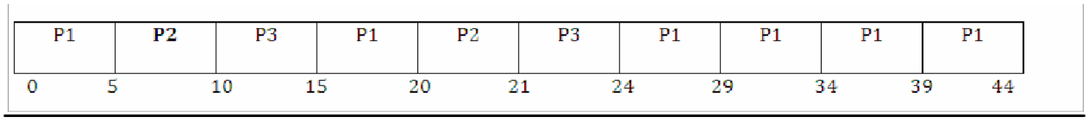
Finish Time for P2 => 15-2 = 13
Finish Time for P3 => 8-4 =4 Finish
Time for P4 => 20-6 = 14 Finish Time
for P5 => 10-8 = 2

Average Turn around time => 36/5 = 7.2 ms.

4)ROUND ROBIN SCHEDULING ALGORITHM :

It is designed especially for time sharing systems. Here CPU switches between the processes. When the time quantum expired, the CPU switched to another job. A small unit of time, called a time quantum or time slice. A time quantum is generally from 10 to 100 ms. The time quantum is generally depending on OS. Here ready queue is a circular queue. CPU scheduler picks the first process from ready queue, sets timer to interrupt after one time quantum and dispatches the process.

PROCESS	BURST TIME
P1	30
P2	6
P3	8



AVERAGE WAITING TIME :

Waiting time for P1 => 0+(15-5)+(24-20) => 0+10+4 = 14
Waiting time for P2 => 5+(20-10) => 5+10 =
15 Waiting time for P3 => 10+(21-15) => 10+6
= 16 Average waiting time => (14+15+16)/3 =
15 ms.

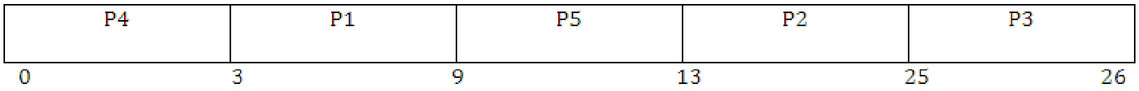
AVERAGE TURN AROUND TIME :

FORMULA : Turn around time = waiting time + burst
TimeTurn around time for P1 => 14+30 =44
Turn around time for P2 => 15+6 =
21 Turn around time for P3 => 16+8
= 24
Average turn around time => (44+21+24)/3 = 29.66 ms

5) PRIORITY SCHEDULING :

PROCESS	BURST TIME	PRIORITY
P1	6	2
P2	12	4
P3	1	5
P4	3	1
P5	4	3

P4 has the highest priority. Allocate the CPU to process P4 first next P1, P5, P2, P3.



AVERAGE WAITING TIME :

Waiting time for P1 => 3-0 =3
Waiting time for P2 => 13-0 = 13
Waiting time for P3 => 25-0 = 25
Waiting time for P4 => 0
Waiting time for P5 => 9-0 =9

Average waiting time => (3+13+25+0+9)/5 = 10 ms

AVERAGE TURN AROUND TIME :

Turn around time for P1 =>3+6 = 9
Turn around time for P2 => 13+12=
25Turn around time for P3 => 25+1
= 26 Turn around time for P4 => 0+3=
3 Turn around time for P5 => 9+4 =
13

Average Turn around time => (9+25+26+3+13)/5 = 15.2 ms

Disadvantage: Starvation

Starvation means only high priority process are executing, but low priorityprocess are waiting for the CPU for the longest period of the time.

Multiple – processor scheduling:

When multiple processes are available, then the scheduling gets more complicated, because there is more than one CPU which must be kept busy and in effective use at all times.

Load sharing resolves around balancing the load between multiple processors. Multi processor systems may be heterogeneous (It contains different kinds of CPU's) (or) Homogeneous(all the same kind of CPU).

1) Approaches to multiple-processor schedulinga)Asymmetric multiprocessing

One processor is the master, controlling all activities and running all kernel code,while the other runs only user code.

b)Symmetric multiprocessing:

Each processor schedules its own job. Each processor may have its own private queue of ready processes**.**

2) Processor Affinity

Successive memory accesses by the process are often satisfied in cache memory. what happens if the process migrates to another processor. the contents of cache memory must be invalidated for the first processor, cache for the second processor must be repopulated. Most Symmetric multi processor systems try to avoid migration of processes from one processor to another processor, keep a process running on the same processor. This is called processor affinity.

a) Soft affinity:

Soft affinity occurs when the system attempts to keep processes on the same processor but makes no guarantees.

b) Hard affinity:

Process specifies that it is not to be moved between processors.

3) Load balancing:

One processor wont be sitting idle while another is overloaded. Balancing can be achived through push migration or pull migration.

Push migration:

Push migration involves a separate process that runs periodically(e.g every 200 ms) and moves processes from heavily loaded processors onto less loaded processors. **Pull migration:**

Pull migration involves idle processors taking processes fromthe readyqueues of the otherprocessors.

UNIT-III

Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, DeadlockPrevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock.

Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors

DEADLOCKS

System model:

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types, each consisting of some number of identical instances. Memory space, CPU cycles, files, I/O devices are examples of resource types. If a system has 2 CPUs, then the resource type CPU has 2 instances.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its task. The number of resources as it requires to carry out its task. The number of resources requested may not exceed the total number of resources available in the system. A process cannot request 3 printers if the system has only two.

A process may utilize a resource in the following sequence:

(I) REQUEST: The process requests the resource. If the request cannot be granted immediately (if the resource is being used by another process), then therequesting process must wait until it can acquire theresource.

(II) USE: The process can operate on the resource .if the resource is a printer, the process can print on theprinter.

(III) RELEASE: The process release theresource.

For each use of a kernel managed by a process the operating system checks that the process has requested and has been allocated the resource. A system table records whether each resource is free (or) allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

To illustrate a deadlocked state, consider a system with 3 CDRW drives. Each of 3 processes holds one of these CDRW drives. If each process now requests another drive, the 3 processes will be in a deadlocked state. Each is waiting for the event “CDRW is released” which can be caused only by one ofthe other waiting processes. This example illustrates a deadlock involving the same resourcetype.

Deadlocks may also involve different resource types. Consider a system with one printer and one DVD drive. The process Pi is holding the DVD and process Pj is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

DEADLOCK CHARACTERIZATION:

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

NECESSARY CONDITIONS:

A deadlock situation can arise if the following 4 conditions hold simultaneously in a system:

1. MUTUAL EXCLUSION: Only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until theresource has beenreleased.
2. HOLD AND WAIT: A process must be holding at least one resource and waitingto acquire additional resources that are currently being held by otherprocesses.
3. NO PREEMPTION: Resources cannot be preempted. A resource can be released only voluntarily by the process holding it, after that process has completed itstask.
4. CIRCULAR WAIT: A set {P0,P1,…..Pn} of waiting processes must exist such that P0 is waiting for resource held by P1, P1 is waiting for a resource held by P2,……,Pn-1 is waiting for a resource held by Pn and Pn is waiting for a resource held byP0.

RESOURCE ALLOCATION GRAPH

Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph. This graph consists of a set of vertices V and a set of edges E. the set of verticesV is partitioned into 2 different types of nodes:

P = {P1, P2….Pn}, the set consisting of all the active processes in the system. R= {R 1, R2….Rm}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi ->Rj. It signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.

A directed edge from resource type Rj to process Pi is denoted by Rj ->Pi, it signifies that an instance of resource type Rj has been allocated to process Pi.

A directed edge Pi ->Rj is called a requested edge. A directed edge Rj->Piis called an assignmentedge.

We represent each process Pi as a circle, each resource type Rj as a rectangle. Since resource type Rj may have more than one instance. We represent each such instance as a dot within the rectangle. A request edge points to only the rectangle Rj. An assignment edge must also designate one of the dots in therectangle.

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

The sets P, R, E:

P= {P1, P2, P3}
R= {R1, R2, R3, R4}
E= {P1 ->R1, P2 ->R3, R1 ->P2, R2 ->P2, R2 ->P1, R3 ->P3}

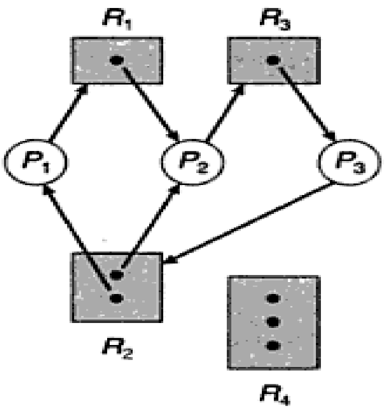


Figure Resource-allocation graph with a deadlock.

One instance ofresource type R1

Two instances ofresource type R2 One instance of resource type R3 Three instances of resource type R4 **PROCESS STATES:**

Process P1 is holding an instance of resource type R2 and is waiting for an instance of resourcetype R1.

Process P2 is holding an instance of R1 and an instance of R2 and is waiting for instance of R3.Process P3 is holding an instance of R3.

If the graph contains no cycles, then no process in the system is deadlocked. Ifthe graph does contain a cycle, then a deadlock may exist.

Suppose that process P3 requests an instance of resource type R2. Since no resource instance iscurrently available, a request edge P3 ->R2 is added to the graph.

2 cycles:

P1 ->R1 ->P2 ->R3 ->P3 ->R2 ->P1P2 ->R3 ->P3 ->R2 ->P2

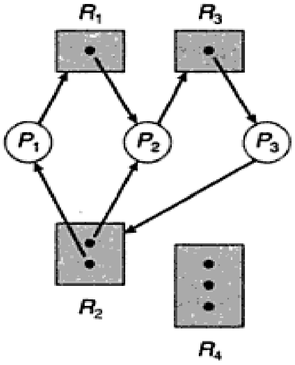


Figure Resource-allocation graph with a deadlock.

Processes P1, P2, P3 are deadlocked. Process P2 is waiting for the resource R3, which is held byprocess P3.process P3 is waiting for either process P1 (or) P2 to release resource R2. In addition,process P1 is waiting for process P2 to release resource R1.

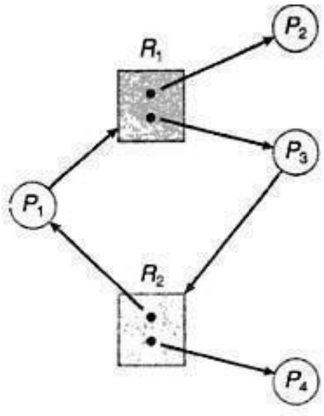


Figure Resource-allocation graph with a cycle but no deadlock.

We also have a cycle: P1 ->R1 ->P3 ->R2 ->P1

However there is no deadlock. Process P4 may release its instance of resource type R2. Thatresource can then be allocated to P3, breaking the cycle.

DEADLOCK PREVENTION

For a deadlock to occur, each of the 4 necessary conditions must held. By ensuring that at leastone ofthese conditions cannot hold, we can prevent the occurrence of a deadlock.

Mutual Exclusion – not required for sharable resources; must hold for non sharable resources

Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources

o Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process hasnone

o Low resource utilization; starvation possible

No Preemption –

o If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released

o Preempted resources are added to the list of resources for whichthe process is waiting

o Process will be restarted only when it can regain its old resources, aswell as the new ones that it is requesting

Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration **Deadlock Avoidance**

Requires that the system has some additional *a priori* information available

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes .

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

System is in **safe state** if there exists a sequence <*P1, P2, ..., Pn*> of ALL the processes in the systems such that for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the *Pj*, with *j* <*I*

That is:

o If Pi resource needs are not immediately available, then *Pi* can wait until all *Pj* have finished

o When *Pj* is finished, *Pi* can obtain needed resources, execute,return allocated resources, and terminate

o When *Pi* terminates, *Pi* +1 can obtain its needed resources, and so on If a system is in safe state no deadlocks

If a system is in unsafe state possibility of deadlock Avoidance ensure that a system will never enter an unsafe state**Avoidance algorithms**

Single instance ofa resource type

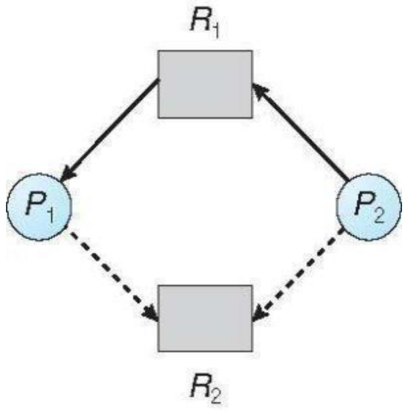
o Use a resource-allocation graph Multiple instances ofa resource type

o Use the banker's algorithm

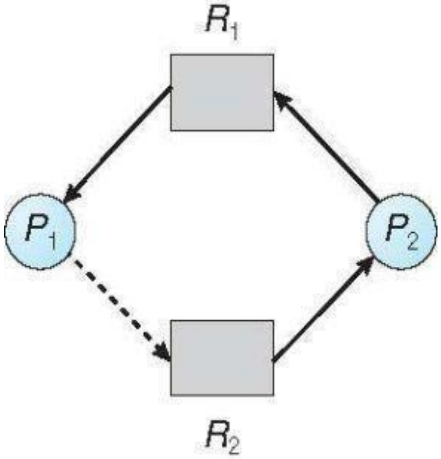
Resource-Allocation Graph Scheme

Claim edge*PiÆRj* indicated that process *Pj* may request resource *Rj*;represented by a dashed line

Claim edge converts to request edge when a process requests a resource Request edge converted to an assignment edge when the resource is allocatedto the process When a resource is released by a process, assignment edge reconverts to a claim edge Resources must be claimed *a priori* in the system



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

Multiple instances

Each process must a priori claim maximum use

When a process requests a resource it may have to wait

When a process gets all its resources it must return them in a finite amount of time Let *n* = number of processes, and *m* = number of resources types.

Available*:* Vector of length *m*. Ifavailable [*j*] = *k*, there are *k* instances ofresource type *Rj*available

Max*: n x m* matrix. If *Max* [*i*,*j*] = *k*, then process *Pi*may request at most *k* instances ofresource type *Rj*

Allocation*: n* x *m* matrix. If Allocation[*i*,*j*] = *k* then *Pi* is currentlyallocated *k* instances of *Rj*

Need*: n* x *m* matrix. If *Need*[*i*,*j*] = *k*, then *Pi* may need *k* more instances of *Rj*to complete its task

Need [*i*,*j*] = *Max*[*i*,*j*] – *Allocation* [*i*,*j*]

Safety Algorithm

1. Let Work and Finish be vectors of length mand n,respectively.
2. Initialize: Work = Available

Finish [i] = false fori = 0, 1, …,n- 1

3. Find an isuch that both:

(a) Finish [i] = false

(b) Needi=Work

If no such iexists, go to step 4

4. Work = Work + AllocationiFinish[i] = true

go to step 2

5. IfFinish [i] == true for all i, then the system is in a safe state

Resource-Request Algorithm for Process *Pi*

Request = request vector for process *Pi*. If *Requesti*[*j*] = *k* then process *Pi* wants *k* instances of resource type *Rj*

1. If *Requesti*£*Needi*go to step 2. Otherwise,

raise error condition, since processhas exceeded its maximum claim

2. If *Requesti*£*Available*, go to step 3. Otherwise

Pi must wait, since resources are not available

3. Pretend to allocate requested resources to *Pi* by modifying the state as follows:

Available = Available –
Request; Allocationi=
Alloc
ationi + Requesti;Needi=Needi – Requesti;

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

Example of Banker's Algorithm(REFER CLASS NOTES)

consider 5 processes *P*0 through

*P*4; 3 resourcetypes:

A (10 instances), *B* (5instances), and *C* (7 instances)

Snapshot at time *T*0:

Allocation	*Max*	*Available*
A B C	*A B C*	*A B C*
*P*0 0 1 0	7 5 3	3 3 2
*P*1 2 0 0	3 2 2	
*P*2 3 0 2	9 0 2	
*P*3 2 1 1	2 2 2	
*P*4 0 0 2	4 3 3	

Σ The content of the matrix *Need* is defined to be *Max*

– Allocation NeedA B C

The system is in a safe state since the sequence <*P*1, *P*3, *P*4, *P*2, *P*0>

satisfies safety criteria

***P*1 Request (1,0,2)**

Check that Request £ Available (that i. s, (1,0,2) £ (3,3,2) true

Allocation	*Need*	*Available*
A B C	*A B C*	*A B C*
*P*0 0 1 0	7 4 3	2 3 0
*P*1 3 0 2	0 2 0	
*P*2 3 0 2	6 0 0	
*P*3 2 1 1	0 1 1	
*P*4 0 0 2	4 3 1	

Executing safety algorithm shows that sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies safety requirement

Deadlock Detection

Allow system to enter deadlock stateDetection algorithm

Recovery scheme

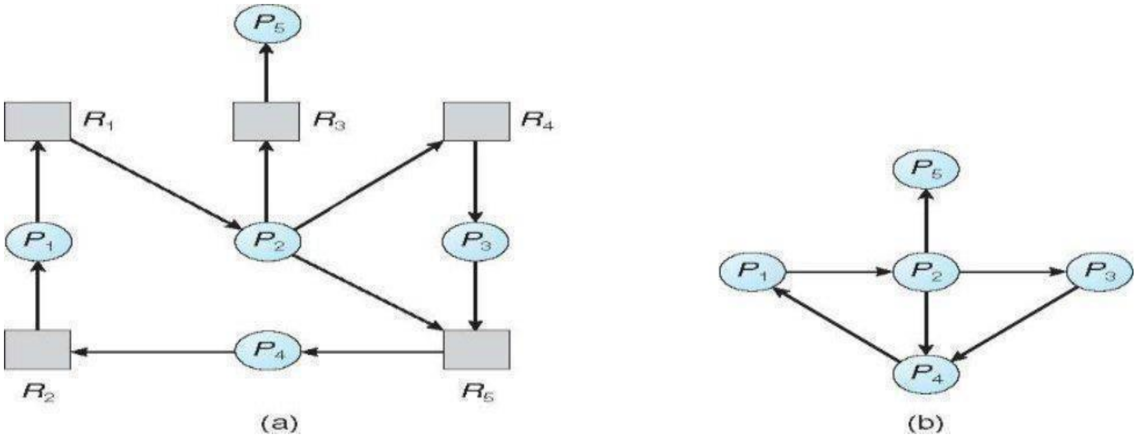
Single Instance of Each Resource Type

Maintain *wait-for* graphNodes are processes *PiÆP*
*j*if *Pi*is waiting for*Pj*

Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle,there exists a deadlock

An algorithm to detect a cycle in a graph requires an order of *n*2 operations,where *n* is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

Available*:* A vector of length *m* indicates the number of available resources of each type.

Allocation*:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

Request*:* An *n* x *m* matrix indicates the current request of each process.

If *Request* [*i*][*j*] = *k*, then process *Pi* is requesting *k* more instances ofresource type.*Rj*.

Detection Algorithm

Let Work and Finish be vectors of length m and n, respectively Initialize:

(a) Work = Available

(b) For i = 1,2, …, n, if Allocationiπ 0, then Finish[i] = false; otherwise, Finish[i] = true

2. Find an index isuch that both:

(a) Finish[i] == false

(b) Requesti£Work

If no such i exists, go to step 4

3. *Work* = *Work* + *Allocationi* *Finish*[*i*] = *true*

go to step 2

4. If *Finish*[*i*] == false, for some *i*, 1 £*i*£*n*, then the system is in deadlock

state. Moreover, if

Finish[*i*] == *false*, then *Pi* is deadlocked

Recovery from Deadlock:

Process Termination

Abort all deadlocked processes

Abort one process at a time until the deadlock cycle is eliminated In which order should we choose to abort?

- ○ Priority of the process
- ○ How long process has computed, and how much longer to completion
- ○ Resources the process has used
- ○ Resources process needs to complete
- ○ How many processes will need to be terminated
- ○ Is process interactive or batch?

Resource Preemption

Selecting a victim – minimize cost

Rollback – return to some safe state, restart process for that state Starvation – same process may always be picked as victim, include numberof rollback in cost factor

Process Management And Synchronization:

In a single processor multiprogramming system the processor switches between thevarious jobs until to finish the execution of all jobs. These jobs will share the processor time to get the simultaneous execution. Here the overhead is involved in switching back and forth between processes.

Critical Section Problem

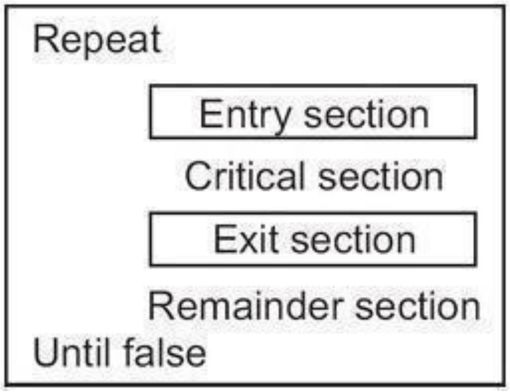
Consider a system consisting of n processes {P0, P1, …, Pn - 1}. Each process hasa segment of code called critical section. In which the process may be changing common variables. Updating a table, writing a file, etc., when one process is executing in its critical section, no other process is allowed into its critical section.

Design a protocol in such a way that the processes can cooperate each other.

Requirements:-

- Mutual exclusion - Only one process can execute their critical sections at any time.
- Progress - If no process is executing in its critical section, any other process can enter based on the selection.
- Bounded waiting - Bounded waiting bounds on the number of times that the other processes are allowed to enter their critical sections after a process has made a request to enter into its critical section and before that the request is granted.

General structure of a process:



Synchronization With Hardware

Certain features of the hardware can make programming task easier and it will alsoimprove system efficiency in solving critical-section problem. For achieving this in uniprocessor environment we need to disable the interrupts when a shared variable is being modified. Whereas in multiprocessor environment disabling the interrupts isa time consuming process and system efficiency also decreases.

Syntax for interrupt disabling process

```
repeat
disable interrupts;
critical section;
enable interrupts;
remainder section&gt;
forever.
```

Code:

Special Hardware Instructions

For eliminating the problems in interrupt disabling techniques particularly in multiprocessor environment we need to use special hardware instructions. In a multiprocessor environment, several processors share access to a common main memory. Processors behave independently. There is no interrupt mechanism b etween processors. At a hardware level access to a memory location, excludes anyother access to that same location. With this foundation designers have proposed several machine instructions that carry out two actions automatically such as reading and writing or reading and testing of a single memory location.
Most widely implemented instructions are:

- Test-and-set instruction
- Exchange instruction

Test-and-set Instruction

Test-and-set instruction executes automatically. If two test-and-set instructions are executed simultaneously, each on a different CPU, then they will execute sequentially. That is access to a memory location excludes any other access to thatsame location which is shared one.

Implementation

Code:

```
function Test-and-set (var i : integer) : boolean;
begin
 if i = 0 then
begin
 i : = 1;
Test-and-set := true
end
else
 Test-and-set: = false
 end.
```

Exchange Instruction

A global boolean variable lock is declared and is initialized to false.

Code:

```
Var waiting : array [0 ... n - 1] of boolean
lock : boolean
 Procedure Exchange (Var a, b : boolean);
var temp:boolean;
begin  temp: = a;
 a : = b;
 b: = temp;
```

```
end;
repeat
 key:= true;
repeat
Exchange (lock, key);
until key = false;
 critical section
 lock: = false;
 remainder section
until false;
```

Properties of the Machine-instruction Approach

Advantages

- It is applicable to any number of processes on either a single processor or multipleprocessors which are sharing main memory.
- It is simple and easy to verify.
- Support multiple critical sections.

Disadvantages

- Busy-waiting is employed.
- Starvation is possible, because selection of a waiting process is arbitrary.
- Deadlock situation may rise.

Semaphores

The above solution to critical section problem is not easy to generalize to more complex problems. For overcoming this problem a synchronization tool called a‘semaphore’ is used. A semaphore ‘S’ is an integer variable. It is accessed only
throughtwo standard atomic operations ‘wait’ and ‘signal’. These operations can betermed as P and V.

Principle

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

Operations

- A semaphore may be initialized to a non-negative value.
- The ‘wait’ operation decrements the semaphore value. If the value becomes negative, then the process executing the wait is blocked.

- The signal operation increments the semaphore value. If the value is not positive,then a process blocked by a wait operation is unblocked.

No-op stands for no operation.

Code:

```
wait (s) : while S = 0 do no-op
 S: = S - 1;
Signal (s) : S : = S + 1;
```

Usage

To deal with the n-process critical-section problem ‘n’ processes share a semaphore by initializing mutex to 1.

Code:

```
repeat
 wait (mutex);
 critical section
 signal (mutex);
remainder section
until false
```

Another usage of semaphores is to solve various synchronization problems. For example, concurrently running processes

Code:

```
 P1 with a statement S1.
S1;
Signal (synch);
 and P2 with a statement S2
wait (synch);
 S2;
```

By initializing synch to zero (0), execute S2 only after P1 has invoked signal (synch),which is after S1.

Implementation

In the above semaphore definition the waiting process trying to enter its critical section must loop continuously in the entry code.This continuous looping in the entrycode is called busy waiting.Busy waiting wastes CPU cycles, this type of semaphoreis called spinlock. These spinlocks are useful in multiprocessor systems. No context switch is required when a process waits on a lock. Context switch may take considerable time. Thus when locks are expected to be held for short times, spinlocks are useful.

When a process executes wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself, and restart by wakeup when some other process executes a signal operation.That is wakeup operation changes the process from waiting to ready.

Binary Semaphore

Earlier semaphore is known as counting semaphore, since its integer value can range over an unrestricted domain, whereas a Binary semaphore value can range between 0 and 1. A binary semaphore is simpler to implement than counting semaphore. We implement counting semaphore (s) in terms of binary semaphores with the following data structures.

Code:

```
Var S1: binary-semaphore;
 S2: binary-semaphore;
S3: binary-semaphore;
 C: integer;
```

initially S1 = S3 = 1, S2 = 0

value of C is the initial value of the counting semaphore ‘S’.

Wait operation

Code:

```
 wait (S3);
wait (S1);
 C: = C - 1;
 if C < 0 then
begin
 signal (S1);
wait (S2);
end
else
signal (S1);
 signal (S3);
```

Signal operation

Code:

```
wait (S1);
 C: = C + 1;
 if C = 0 then
signal (S2);
 signal (S1);
```

Classical Problems Of Synchronization

For solving these problems use Semaphores concepts. In these problems communication is to takes place between processes and is called ‘Inter ProcessCommunication (IPC)’.

1) Producer-Consumer Problem

Producer-consumer problem is also called as Bounded-Buffer problem. Pool consists of n buffers, each capable of holding one item. Mutex semaphore is initialized to ‘1’. Empty and full semaphores count the number of empty and full

buffers, respectively. Empty is initialized to ‘n’ and full is initialized to ‘0’ (zero). Here one or more producers are generating some type of data and placing these in a buffer. A single consumer is taking items out of the buffer one at a time. It prevents the overlap of buffer operations.

Producer Process

Code:

```
repeat
    ...
   produce an item in next P
   ...
   wait (empty);
   wait (mutex);
   ...
   add next P to buffer
   ...
   signal (mutex);
   signal (full);
until false;
```

Consumer Process

Code:

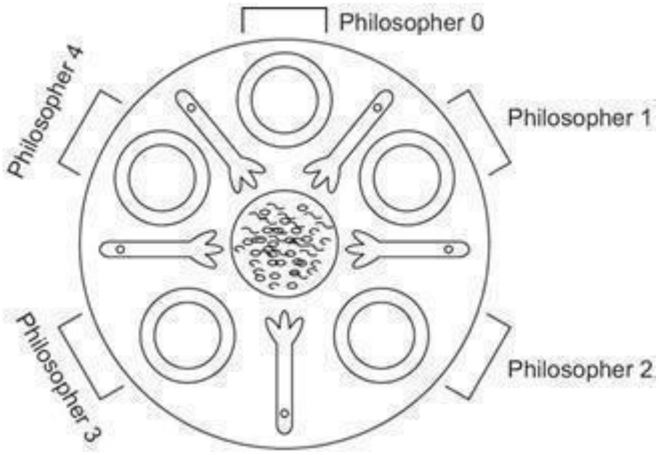
```
repeat
```

```
          wait (full);
          wait (mutex);
          ...
          remove an item from buffer to next C
          ...
          signal (mutex);
          signal (empty);
          ...
          Consume the item in next C
          ...
until false;
```

With these codes producer producing full buffers for the consumer. Consumer producing empty buffers for the producer.

2) Dining-Philosophers Problem:-

Dining-pilosophers problem is posed by Disjkstra in 1965. This problem is very simple. Five philosophers are seated around a circular table. The life of each philosopher consists of thinking and eating. For eating five plates are there, one for each philosopher. There is a big serving bowl present in the middle of the table with enough food in it. Only five forks are available on the whole. Each philosopher needs to use two forks on either side of the plate to eat food.



Now the problem is algorithm must satisfy mutual exclusion (i.e., no two philosohers can use the same fork at the same time) deadlock and starvation.

For this problem various solutions are available.

1. Each philosopher picks up first the fork on the left and then the fork on the right. After eating two forks are replaced on the table. In this case if all the philosophers are hungry all will sit

down and pick up the fork on their left and all reach out for the other fork, which is not there. In this undegnified position, all philosophers will starve.It is the deadlock situation.

2. Buy five additional forks
3. Eat with only one fork
4. Allow only four philosophers at a time, due to this restriction at least one philosopherwill have two forks. He will eat and then replace, then there replaced forks are utilized by the other philosophers (washing of the forks is implicit).
5. Allow philosopher to pick up the two forks if both are available.
6. An asymmetric solution is, an odd philosopher picks up first their left fork and then their right side fork whereas an even philosopher picks up their right side fork first and then their left side fork.

Code for the fourth form of solution is as follows:
Program dining philosophers;
Code:

```
Var fork: array [0 ... 4] of semaphore (: = 1);
 room : semaphore (: = 4);
        i : integer;
procedure philosopher (i : integer);
begin
  repeat
         think;
         wait (room);
         wait (fork [i]);
         wait (fork [(i + 1) mod 5]);
         eat;
         signal (fork [(i + 1) mod 5]);
         signal (fork [i])
         signal (room)
  forever
end;
```

```
begin
  philosopher (0);
  philosopher (1);
  philosopher (2);
  philosopher (3);
  philosopher (4);
end.
```

3) **Readers and Writers Problem**:-

A data object (i.e., file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (i.e., read and write) the sharedobject.

In this context if two readers access the shared data object simultaneously then no problem at all. If a writer and some other process (either reader or writer) access theshared object simultaneously then conflict will raise.

For solving these problems various solutions are there:-

1. Assign higher priorities to the reader processes, as compared to the writerprocesses.
2. The writers have exclusive access to the shared object.
3. No reader should wait for other readers to finish. Here the problem is writers maystarve.
4. If a writer is waiting to access the object, no new readers may start reading. Herereaders may starve.

General structure of a writer process

Code:

```
wait (wrt);
 ...
 writing is performed
 ...
signal (wrt);
```

General structure ofa reader process

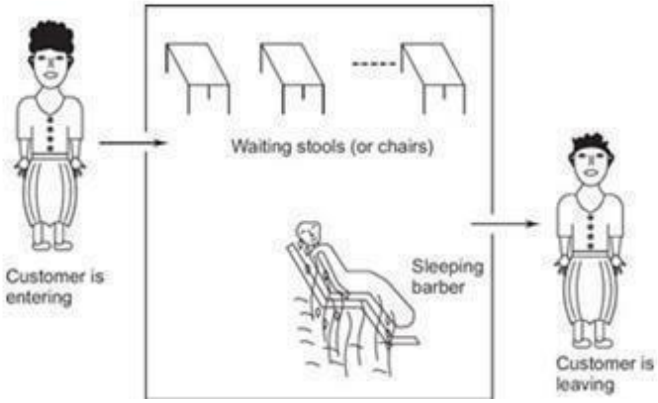
Code:

```
wait (mutex);
  readcount: = readcount + 1;
  if readcount = 1 then wait (wrt);
signal (mutex);
  ...
  reading is performed
  ...
wait (mutex);
  read count: = readcount - 1;
  if readcount = 0 then signal (wrt);
signal (mutex);
```

3) **Sleeping Barber Problem**:-

A barber shop has one barber chair for the customer being served currently and few chairs for the waiting customers (if any). The barber manages his time efficiently.

1. When there are no customers, the barber goes to sleep onthe barber chair.
2. As soon as a customer arrives, he has to wake up the sleeping barber, and ask for ahair cut.
3. If more customers arrive whilst the barber is serving a customer, they either sit downin the waiting chairs or can simply leave the shop.



For solving this problem define three semaphores

1. Customers — specifies the number of waiting customers only.
2. Barber — 0 means the barber is free, 1 means he is busy.
3. Mutex — mutual exclusion variable.

Code:

```
# Define Chairs 4 typedef
int  semaphore;  semaphore
customers  =  0;  semaphore
barber = 0; semaphore mutex
= 1;
int waiting = 0; Void
barber (Void)
{
        while (TRUE)
        {
                waiting = waiting - 1;
                signal (barber);
                cut-hair();
        }
}
Void customer (void)
{
        if (waiting < chairs)
```

```
{
        waiting  =  waiting  +  1;
        signal  (customers);  get-
        hair cut();
}
else
{
        wait (mutex);
}
```

```
# Define Chairs 4
typedef int semaphore;
semaphore customers = 0;
semaphore barber = 0;
semaphore mutex =  1;
int waiting = 0;
Void barber (Void)
{
        while (TRUE)
        {
                waiting = waiting - 1;
                signal  (barber);
                cut-hair();
        }
}
Void customer (void)
{
        if (waiting < chairs)
        {
                waiting = waiting + 1;
                signal  (customers);
                get-hair cut();
        }
        else
        {
                wait (mutex);
        }
}
```

Critical Regions

Critical regions are high-level synchronization constructs. Although semaphores provide a convenient and effective mechanism for process synchronization, theirincorrect use can still result in timing errors that are difficult to detect, since theseerrors happen only if some particular execution sequences take place, and thesesequences do not always occur.

The critical-region construct guards against certain simple errors associated with thesemaphore solution to the critical-section problem made by a programmer. Critical- regions does not necessarily eliminate all synchronization errors; rather, it reduces them.

All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait (mutex) before entering into the critical section, and signal (mutex) afterward. If this sequence is not observed, two processes may be intheir critical sections simultaneously.

Difficulties

- Suppose that a process interchanges the order wait and signal then several processes may be executing in their critical section simultaneously, violating the mutual exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections. The code is as follows:Signal (mutex);

...

critical section

...

wait (mutex);

- Suppose a process replaces signal (mutex) with wait (mutex) i.e.,wait (mutex);

...

critical section

...

wait (mutex);

Here a deadlock will occur.

- If a process omits the wait (mutex) or the signal (mutex), or both. In this case, eithermutual exclusion is violated or a deadlock will occur.

Solution is use high-level synchronization constructs called critical region and monitor. In these two constructs, assume a process consists of some local data, anda sequential program that can operate on the data. The local data can be accessed by only the sequential program that is encapsulated within the same process.

Processes can however share global data.

Monitors

A monitor is an another high-level synchronization construct. It is an abstraction over semaphores. Coding monitor is similar to programming in high-level programming languages. Monitors are easy to program. The compiler of a programming language usually implements them, thus reducing the scope of programmatic errors.

A monitor is a group or collection of data items, a set of procedures to operate onthe data items and a set of special variables known as 'condition variables'. The condition variables are similar to semaphores. The operations possible on a condition variable are signal and wait just like a semaphore. The main difference between a condition variable and a semaphore is the way in which the signal operation is implemented. Syntax of a monitor is as follows:

Code:

```
monitor monitor name
{ // shared variable declarations
        procedure P1 (...)
        {
                ...
        }
        procedure P2 (...)
        {
                ...
        }
```

```
        Procedure Pn (...)
        {
                ...
        }
        Initialization code (...)
        {
                ...
        }
}
```

The client process cannot access the data items inside a monitor directly. The monitor guards them closely. The processes, which utilize the services of the monitor, need not know about the internal details of the monitors.

At any given time, only one process can be a part of a monitor. For example consider the situation, there is a monitor M having a shared variable V, a procedureR to operate on V and a condition variable C. There are two processes P1 and P2 that want to execute the procedure P. In such a situation the following events will take place:

- Process P1 is scheduled by the operating system
- P1 invokes procedure R to manipulate V.
- While P1 is inside the monitor, the time slice gets exhausted and process P2 isscheduled.
- P2 attempt to invoke R, but it gets blocked as P1 is not yet out of the monitor. As aresult, P2 performs wait operation on C.
- P1 is scheduled again and it exists the monitor and performs signal operation on C.
- Next time P2 is scheduled and it can enter the monitor, invoke the procedure R andmanipulate V.

Monitor concept cannot guarantee that the preceding access sequence will be observed.

Problems

- A process may access a resource without first gaining access permission to theresource.
- A process might never release a resource when once it has been granted access tothe resource
- A process might attempt to release a resource that it never requested.
- A process mayrequest, the same resource twice without first releasing the resource.
- The same difficulties are encountered with the use of semaphores. The possible solution to the current problem is to include the resource access operations within the resource allocator monitor.
- To ensure that the processes observe the appropriate sequences, we must inspect
- all the programs. This inspection is possible for a small, static system, it is notreasonable for a large or dynamic system.

Message Passing

The need for message passing came into the picture because the techniques such as semaphores and monitors work fine in the case of local scope. In other words, aslong as the processes are local (i.e., on the same CPU), these techniques will work fine. But, they are not intended to serve the needs of processes, which are not located on the same machine. For processes which communicate over a network, needs some mechanism to perform the communication with each other, and yet theyare able to ensure concurrency. For eliminating this problem message passing is one solution.

By using the message passing technique, one process (i.e., sender) can safely send a message to another process (i.e., destination). Message passing is similar to remote procedure calls (RPC), the difference is message passing is an operating system concept, whereas RPC is a data communications concept.

Two primitives in message passing are:

- Send (destination, & message); i.e., send call
- receive (source & message); i.e., receive call

UNIT -IV

Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory implementation in linux. Corresponding system calls.

Memory Management and Virtual Memory - Logical versus Physical Address Space, Swapping, Contiguous Allocation, Paging, Segmentation, Segmentation with Paging, Demand Paging, Page Replacement, Page Replacement Algorithms.

Inter Process Communication

- Processes share memory
  - data in shared messages
- Processes exchange messages
  - message passing via sockets
- Requires synchronization
  - mutex, waiting

Inter Process Communication(IPC) is an OS supported mechanism for interaction among processes (coordination and communication)

- Message Passing
  - e.g. sockets, pips, messages, queues

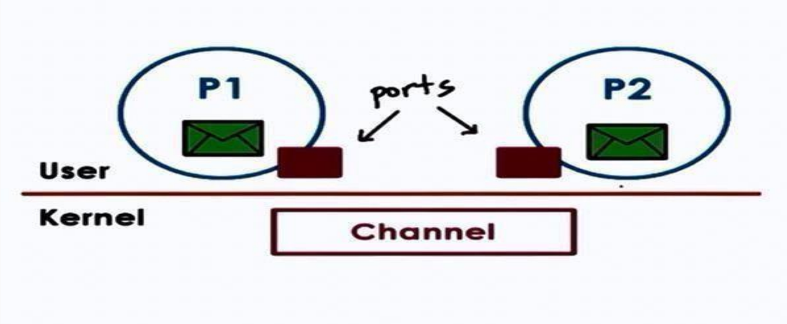
  Memory based IPC
  - shared memory, memory mapped files

  Higher level semantics
  - files, RPC

  Synchronization primitives

Message Passing

- Send/Receive messages
- OS creates and maintains a channel
  - buffer, FIFO queue
- OS provides interfaces to processes
  - a port
  - processes send/write messages to this port
  - processes receive/read messages from this port



- Kernel required to
  - establish communication
  - perform each IPC operation
  - send: system call + data copy
  - receive: system call + data copy

  Request-response:4xuser/kernelcrossings+4x data copies

Advantages

- simplicity : kernel does channel management and synchronization

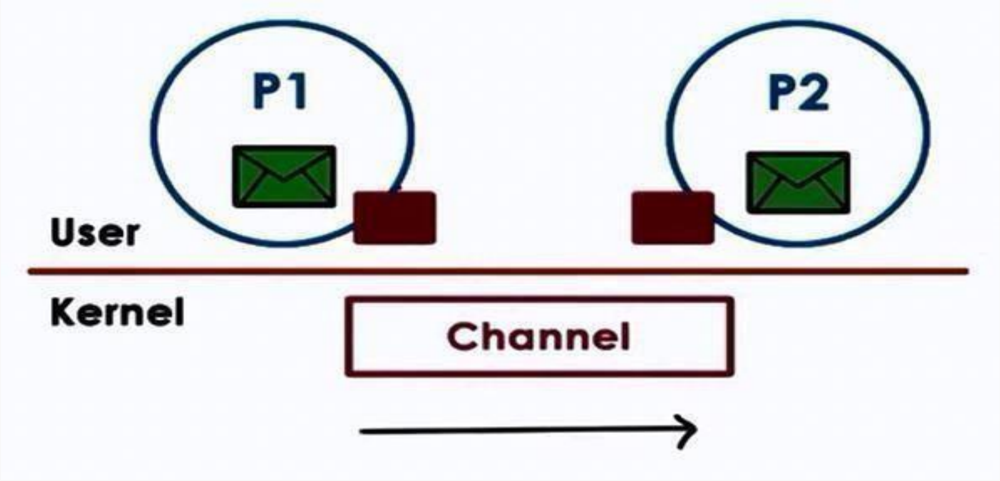
Disadvantages

- Overheads

Forms of Message Passing IPC

1. Pipes

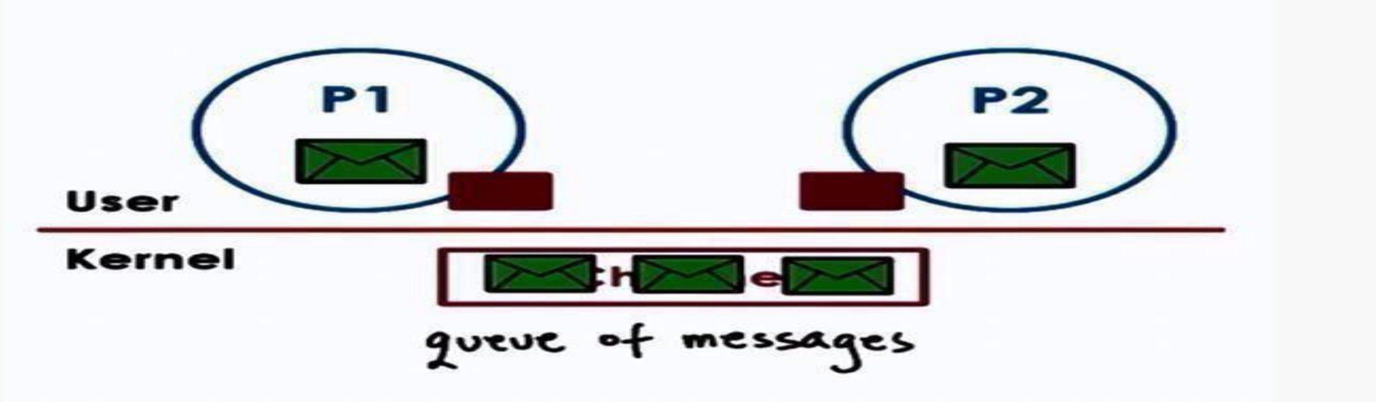
- Carry byte stream between 2 process
- e.g connect output from 1 process to input of another



2. Message queues

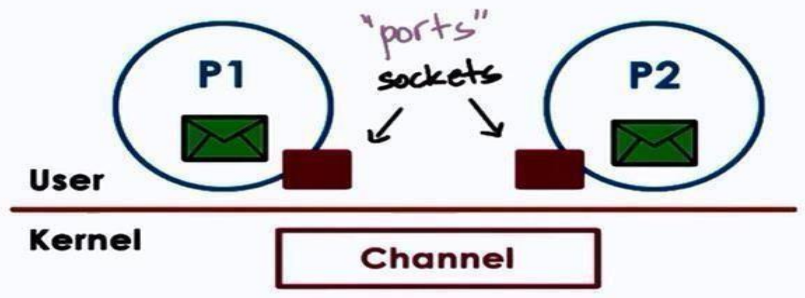
Carry "messages" among processes

- OS management includes priorities, scheduling of message delivery
- APIs : Sys-V and POSIX



3. Sockets

- send() and recv() : pass message buffers
- socket() : create kernel level socket buffer
- associated necessarykernel processing (TCP-IP,..)
- If different machines, channel between processes and network devices
- If same machine, bypass full protocol stack

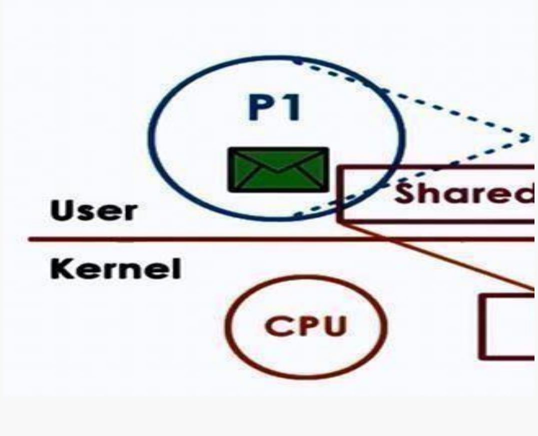


Shared Memory IPC

- read and write to shared memory region
- OS establishes shared channel between the processes

1. physical pages mapped into virtual address space
2. VA(P1) and VA(P2) map to same physical address
3. VA(P1) != VA(P2)
4. physical memory doesn't need to be contiguous

- APIs : SysV, POSIX, memory mapped files, Android ashmem



A vantages

- System calls only for setup data cop es potentially reduced (but not eliminated)

D sdvantages

- explicit synchronization
- communication protocol, shared buffer management
  - programmer's responsibility

O erheads for 1. Message Passing : must perform multiple copies 2. Shared Memory : must establish all mappings among processes' address space and shared memory pages

Cop vs Map
Goal for both is to transfer data from one into target address space

Copy (Message Passing)	Map (Shared Memory)
CPU cycles to copy data to/from port	CPU cycles to map memory into address space
	CPU to copy data to channel
	If channel setup once, use many times (good payoff)
	Can perform well for 1 time use

- Large Data: t(Copy) >> t(Map)
  - e.g. trade-off exercised in Window "Local" Procedure Calls (LPC)

Shared Memory and Synchronization
Use threads accessing shared state in a single addressing space, but for processSynchronizationmethod:

1. mechanism supported by processing threading library(pthreads)
2. OS supported IPC for sync

Either method must coordinate

- no of concurrent access to shared segment
- when data is available and ready for consumption

IPC Synchronization

Message Queues	Semaphores
Implement "mutual exclusion" via send/receive	OS supported synchronization construct
	binary construct (either allow process or not)
	Like mutex, if value = 0, stop; if value = 1, decrement(lock) and proceed

Synchronization

Waiting for other processes, so that they can continue working togethermay repeatedly check to continue

- sync using spinlocks

ay wait for a signal to continue

- sync using mutexes and condition vatiab es

iting hurts performance

- PUs waste cycles for checking; cache effects

Limitation of mutextes and condition varia les:

- Error prone/correctness/ease of use
- u lock wrong mutex, signal wrong condition variable

Lack of expressive power

- helper variables for access or priority control

Low-le l support: hardware atomic instructions

Synchronization constructs:

1. Spinlocks (basic sync construct)

- Spinlock is like a mutex

- mutual exclusion
- lock and unlock(free)
- but, lock == busy => spinning

2. Semaphores

- common sync construct in OS kernels
- like a traffic light: Stop and Go
- like mutex, but more general

Semaphore == integer value

- ○ assigned a max value (positive int) => max count

on try(wait)

- ○ if non-zero, decrement and proceed => counting semaphore

if initialized with 1

- ○ semaphore == mutex(binary semaphore)

on exit(post)

- ○ increment

Syncing different types of accesses

Reader/Writer locks

read (don't modify)	write (always modify)
shared access	exclusive access

- RW locks
  - ○ specify type of access, then lock behaves accordingly

Monitors (highlevel construct)

- shared resource
- entry resource
- possible condition variables

On entry:

- lock, check

  On exit:

- unlock, check, signal

More synchronization constructs

- serializers
- path expressions
- barriers
- rendezvous points
- optimistic wait-free sync (RCU) [Read Copy Update]

All need hardware support.

Need for hardware support

- Problem
  - concurrent check/update on different CPUs can overlap

Atomic instructions

Critical section with hardware supported synchronization

Hardware specific

- Test-and-set
  - returns(tests) original values and sets new-value!= 1 (busy) automatically
  - first thread: test-and-set(lock) => 0 : free
  - next ones: test-and-set(lock) => 1 busy
    - reset lock to 1, but that's okay
  - + : Latency
  - + : minimal (Atomic)
  - + : Delay potentially min
  - - : Contention processors go to memory on each spin - To reduce contention, introduce delay - Static(based on a fixed value) or Dynamic(backoff based, random delay)

read-and-increment

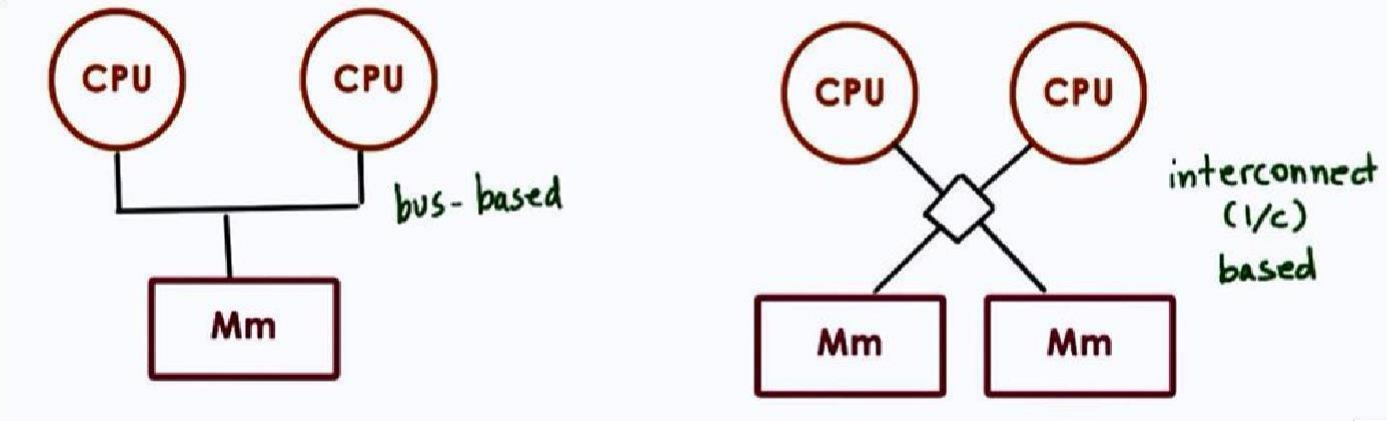
compare-and-swap

Guarantees

- atomicity mutual
- exclusion
- queue all concurrent instructions but one

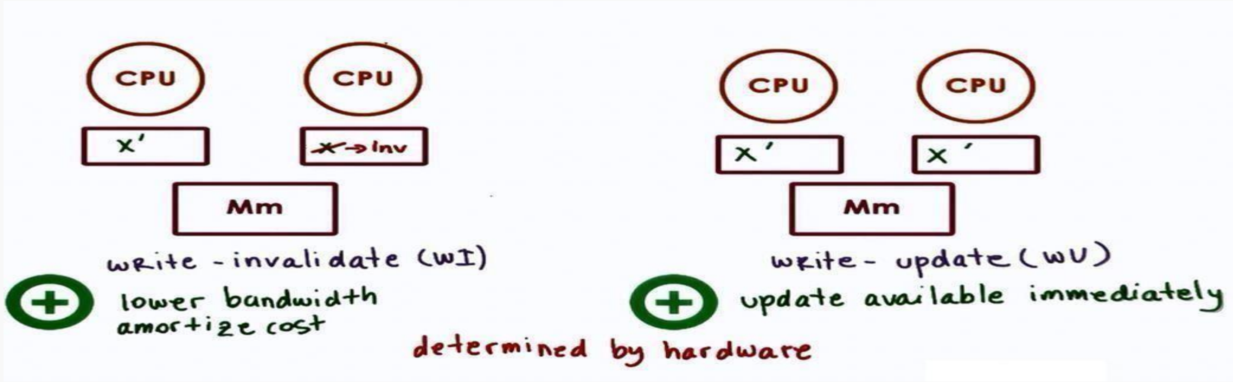
Shared Memory Multiprocessors

Also called symmetric multiprocessors (SMP)



- Caches
  - hide memory latency, "memory" further away due to contention
  - no-write, write-through, write-back

Cache Coherence



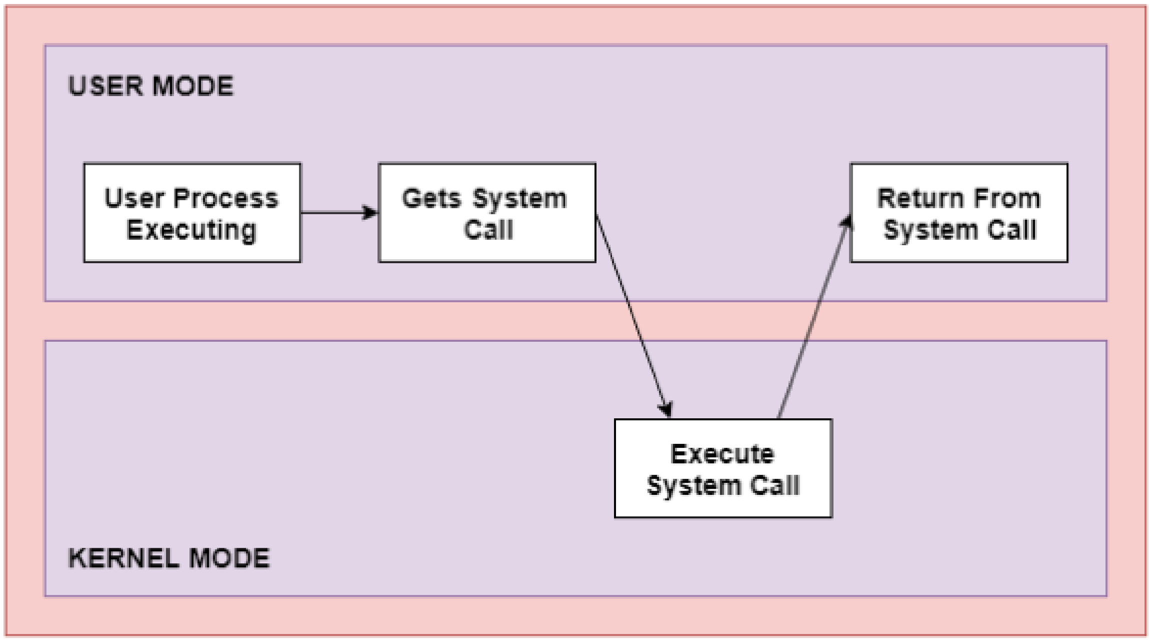
What are system calls in Operating System?

The interface between a process and an operating system is provided by system calls. In general, system calls are

available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource.

Then it requests the kernel to provide the resource via a system call.

A figure representing the execution of the system call is given as follows −



As can be seen from this diagram, the processes execute normally in the user mode until a system call interrupts this. Then the system call is executed on a priority basis in the kernel mode. After the execution of the system call, the control returns to the user mode and execution of user processes can be resumed.

In general, system calls are required in the following situations −

- If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.
- Creation and management of new processes.
- Network connections also require system calls. This includes sending and receiving packets.
- Access to a hardware devices such as a printer, scanner etc. requires a system call.

Types of System Calls

There are mainly five types of system calls. These are explained in detail as follows −

Process Control

These system calls deal with processes such as process creation, process termination etc.

File Management

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

Device Management

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

Information Maintenance

These system calls handle information and its transfer between the operating system and the user program.

Communication

These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

Some of the examples of all the above types of system calls in Windows and Unix are given as follows −

Types of System Calls	**Windows**	**Linux**

Process Control	CreateProcess()
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Management	CreateFile()
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

Device Management	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

Information Maintenance	GetCurrentProcessID()
	SetTimer()
	Sleep()
		getpid()
		alarm()
		sleep()

Communication	CreatePipe()
	CreateFileMapping()
	MapViewOfFile()
		pipe()
		shmget()
		mmap()

There are many different system calls as shown above. Details of some of those system calls are as follows −

open()

The open() system call is used to provide access to a file in a file system. This system call allocates resources to the file and provides a handle that the process uses to refer to the file. A file can be opened by multiple processes at the same time or be restricted to one process. It all depends on the file organisation and file system.

read()

The read() system call is used to access data from a file that is stored in the file system. The file to read can be identified by its file descriptor and it should be opened using open() before it can be read. In general, the read() system calls takes three arguments i.e. the file descriptor, buffer which stores read data and number of bytes to be read from the file.

write()

The write() system calls writes the data from a user buffer into a device such as a file. This system call is one of the ways to output data from a program. In general, the write system calls takes three arguments i.e. file descriptor, pointer to the buffer where data is stored and number of bytes to write from the buffer.

close()

The close() system call is used to terminate access to a file system. Using this system call means that the file is no longer required by the program and so the buffers are flushed, the file metadata is updated and the file resources are de-allocated.

Main Memory

4.1 Background

- Obviously memory accesses and memory management are a very important part of modern computer operation. Every instruction has to be fetched from memory before it can be executed, and most instructions involve retrieving data from memory or storing data in memory or both.
- The advent of multi-tasking OSs compounds the complexity of memory management, because as processes are swapped in and out of the CPU, so must their code and data be swapped in and out of memory, all at high speeds and without interfering with any other processes.
- Shared memory, virtual memory, the classification of memory as read-only versus read-write, and concepts like copy-on-write forking all further complicate the issue.

4.1.1 Basic Hardware

- It should be noted that from the memory chips point of view, all memory accesses are equivalent. The memory hardware doesn't know what a particular part of memory is being used for, nor does it care. This is almost true of the OS as well, although not entirely.
- The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it. (Device drivers communicate with their hardware via interrupts and "memory" accesses, sending short instructions for example to transfer data from the hard drive to a specified location in main memory. The disk controller monitors the bus for such instructions, transfers the data, and then notifies the CPU that the data is there with another interrupt, but the CPU never gets direct access to the disk.)
- Memory accesses to registers are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick.
- Memory accesses to main memory are comparatively slow, and may take a number of clock ticks to complete. This would require intolerable waiting by the CPU if it were not for an intermediary fast memory ***cache*** built into most modern CPUs. The basic idea of the cache is to transfer chunks of memory at a time from the main memory to the cache, and then to access individual memory locations one at a time from the cache.
- User processes must be restricted so that they only access memory locations that "belong" to that particular process. This is usually implemented using a base register and a limit register for each process, as shown in Figures 3.1 and 3.2 below. ***Every*** memory access made by a user process is checked against

these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated. The OS obviously has access to all existing memory locations, as this is necessary to swap users' code and data in and out of memory. It should also be obvious that changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.

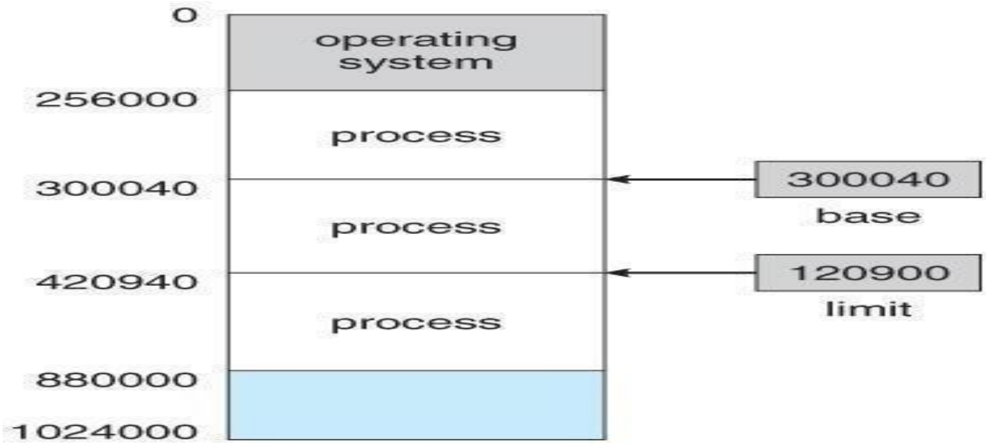


Figure 4.1 - A base and a limit register define a logical addresss space

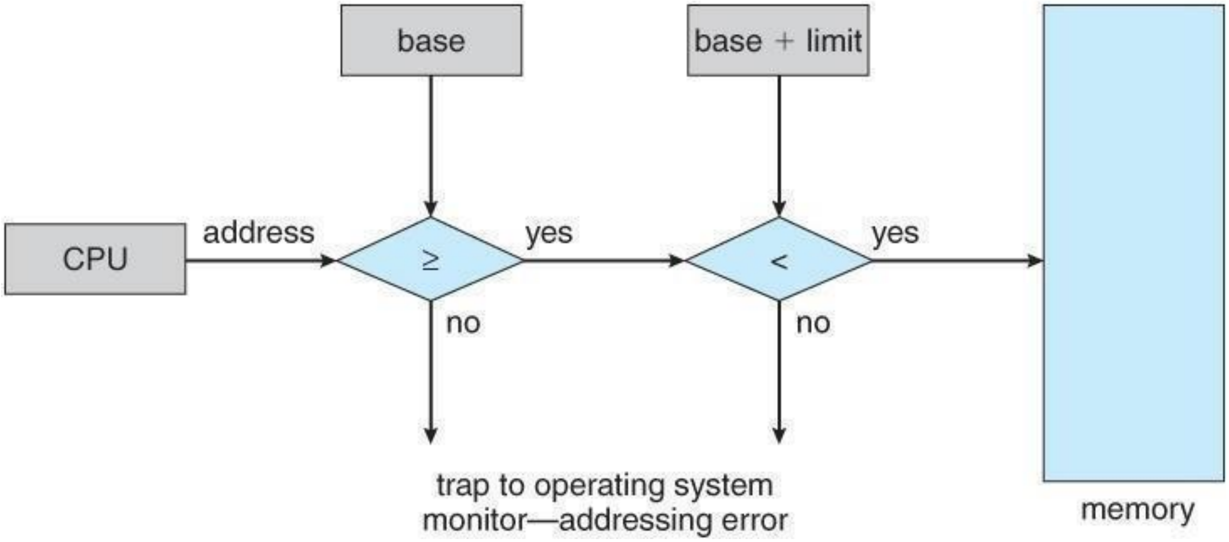


Figure 3.2 - Hardware address protection with base and limit registers

3.1.2 Address Binding

- User programs typically refer to memory addresses with symbolic names such as "i", "count", and "averageTemperature". These symbolic names must be mapped or ***bound*** to physical memory addresses, which typically occurs in several stages:

○ **Compile Time -** If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.

- **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
- **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OSes.

- Figure 3.3 shows the various stages of the binding processes and the units involved in each stage:

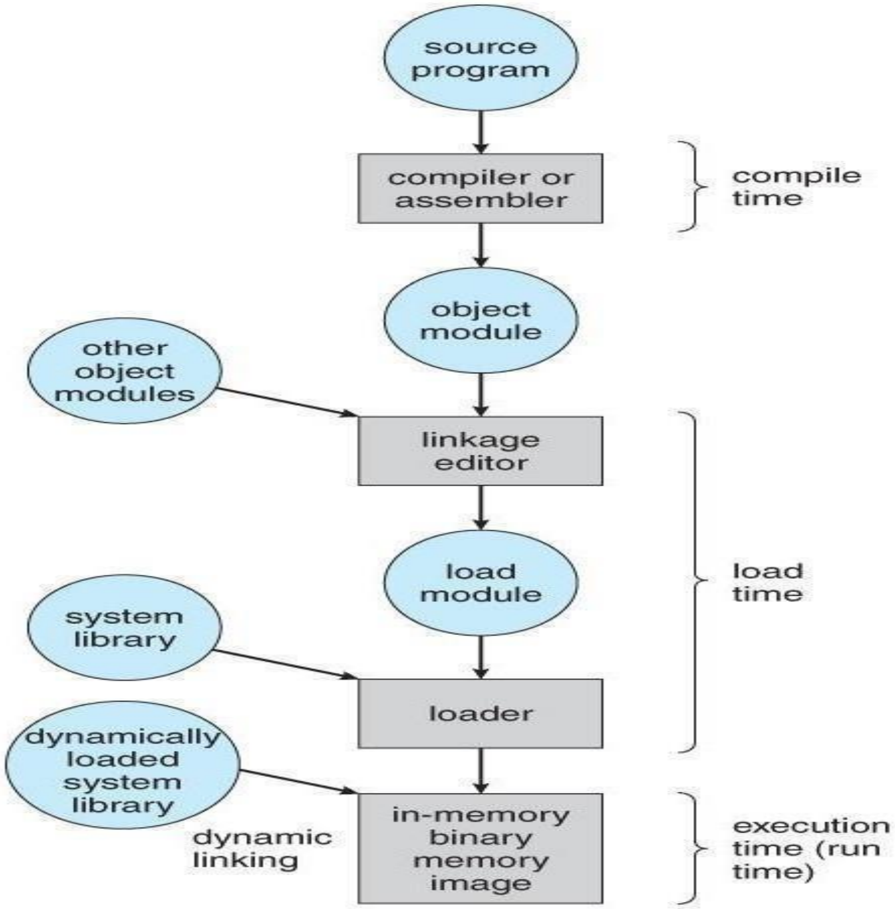


Figure 4.3 - Multistep processing of a user program

4.1.3 Logical Versus Physical Address Space

- The address generated by the CPU is a ***logical address***, whereas the address actually seen by the memory hardware is a ***physical address***.
- Addresses bound at compile time or load time have identical logical and physical addresses.
- Addresses created at execution time, however, have different logical and physical addresses.
  - In this case the logical address is also known as a ***virtual address***, and the two terms are used interchangeably by our text.
  - The set of all logical addresses used by a program composes the ***logical address space***, and the set of all corresponding physical addresses composes the ***physical address space.***
- The run time mapping of logical to physical addresses is handled by the ***memory-management unit, MMU***.
  - The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.

- The base register is now termed a ***relocation register***, whose value is added to every memory request at the hardware level.

- Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.

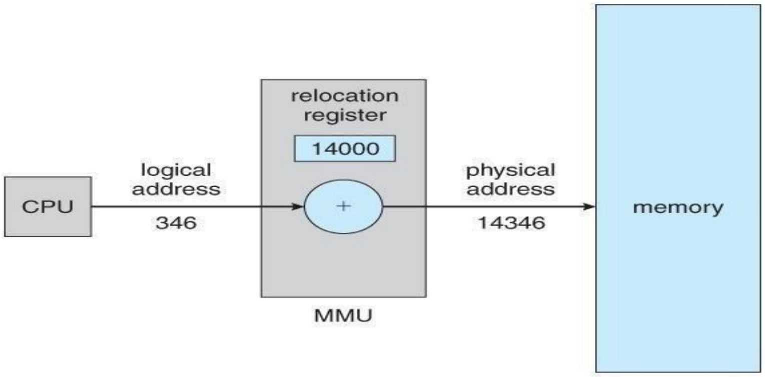


Figure 3.4 - Dynamic relocation using a relocation register

4.1.4 Dynamic Loading

- Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program startup times. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then loading it up if it is not already loaded.

4.1.5 Dynamic Linking and Shared Libraries

- With ***static linking*** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- With ***dynamic linking***, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.
  - This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
  - We will also learn that if the code section of the library routines is ***reentrant***, (meaning it does not modify the code while it runs, making it safe to re-enter it), then main memory can be saved by loading only one copy of dynamically linked routines into memory and sharing the code amongst all processes that are concurrently using it. (Each process would have their own copy of the ***data*** section of the routines, but that may be small relative to the code segments.) Obviously the OS must manage shared routines in memory.
  - An added benefit of ***dynamically linked libraries*** (***DLLs***, also known as ***shared libraries*** or ***shared objects*** on UNIX systems) involves easy upgrades and updates. When a program uses a routine from a standard library and the routine changes, then the program must be re-built (re-linked) in order to incorporate the changes. However if DLLs are used, then as long as the stub doesn't change, the program can be updated merely by loading new versions of the DLLs onto the system. Version information is maintained in both the program and the DLLs, so that a program can specify a particular version of the DLL if necessary.

- In practice, the first time a program calls a DLL routine, the stub will recognize the fact and will replace itself with the actual routine from the DLL library. Further calls to the same routine will access the routine directly and not incur the overhead of the stub access. (Following the UML ***Proxy Pattern***.)

4.2 Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store.***

4.2.1 Standard Swapping

- If compile-time or load-time address binding is used, then processes must be swapped back into the same memory location from which they were swapped out. If execution time binding is used, then the processes can be swapped back into any available location.
- Swapping is a very slow process compared to other operations. For example, if a user process occupied 10 MB and the transfer rate for the backing store were 40 MB per second, then it would take 1/4 second (250 milliseconds) just to do the data transfer. Adding in a latency lag of 8 milliseconds and ignoring head seek time for the moment, and further recognizing that swapping involves moving old data out as well as new data in, the overall transfer time required for this swap is 512 milliseconds, or over half a second. For efficient processor scheduling the CPU time slice should be significantly longer than this lost transfer time.
- To reduce swapping transfer overhead, it is desired to transfer as little information as possible, which requires that the system know how much memory a process ***is*** using, as opposed to how much it ***might*** use. Programmers can help with this by freeing up dynamic memory that they are no longer using.
- It is important to swap processes out of memory only when they are idle, or more to the point, only when there are no pending I/O operations. (Otherwise the pending I/O operation could write into the wrong process's memory space.) The solution is to either swap only totally idle processes, or do I/O operations only into and out of OS buffers, which are then transferred to or from process's main memory as a second step.
- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) However some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again. Windows 3.1 would use a modified version of swapping that was somewhat controlled by the user, swapping process's out if necessary and then only swapping them back in when the user focused on that particular window.

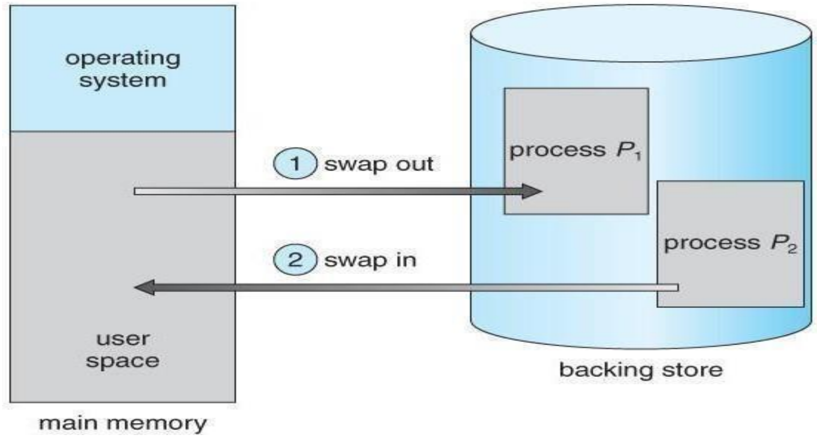


Figure 4.5 - Swapping of two processes using a disk as a backing store

4.3 Contiguous Memory Allocation

- One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is where the interrupt vectors are located, but on older systems part of the OS was loaded high to make more room in low memory (within the 640K barrier) for user processes.)

4.3.1 Memory Protection

- The system shown in Figure 3.6 below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

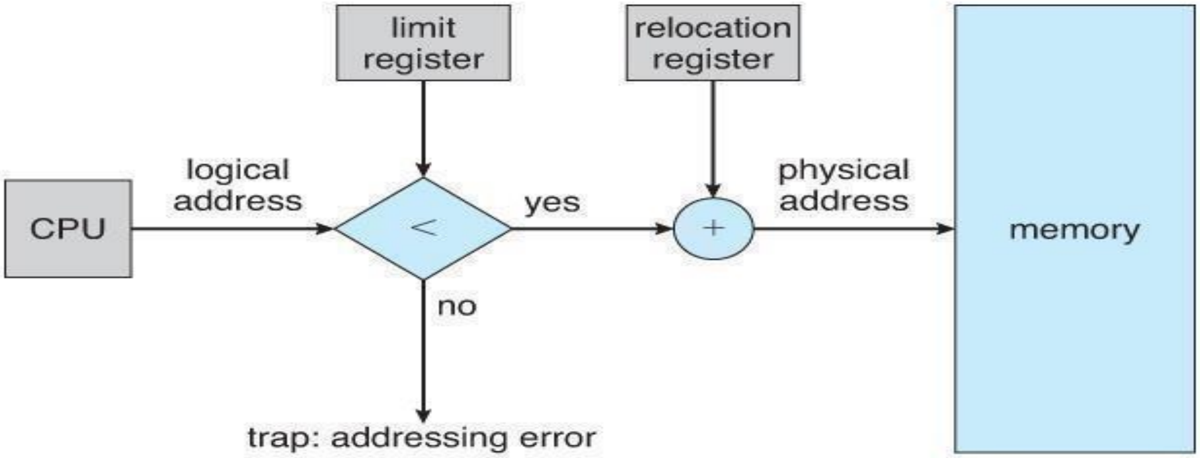


Figure 3.6 - Hardware support for relocation and limit registers

4.3.2 Memory Allocation

- One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:

1. **First fit -** Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
2. **Best fit -** Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be ofany use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
3. **Worst fit -** Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.

- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

4.3.3. Fragmentation

- All the memory allocation strategies suffer from ***external fragmentation***, though first and best fits experience the problems more so than worst fit. External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.
- The amount of memory lost to fragmentation may vary with algorithm, usage patterns, and some design decisions such as which end of a hole to allocate and which end to save on the free list.
- Statistical analysis of first fit, for example, shows that for N blocks of allocated memory, another 0.5 N will be lost to fragmentation.
- ***Internal fragmentation*** also occurs, with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size. For a random distribution of memory requests, on the average 1/2 block will be wasted per memory request, because on the average the last allocated block will be only half full.
  - Note that the same effect happens with hard drives, and that modern hardware gives us increasingly larger drives and memory at the expense of ever larger block sizes, which translates to more memory lost to internal fragmentation.
  - Some systems use variable size blocks to minimize losses due to internal fragmentation.
- If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.
- Another solution as we will see in upcoming sections is to allow processes to use non-contiguous blocks of physical memory, with a separate relocation register for each block.

4.4 Segmentation

4.4.1 Basic Method

- Most users (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a particular use, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.
- For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap, as shown in Figure 4.7:

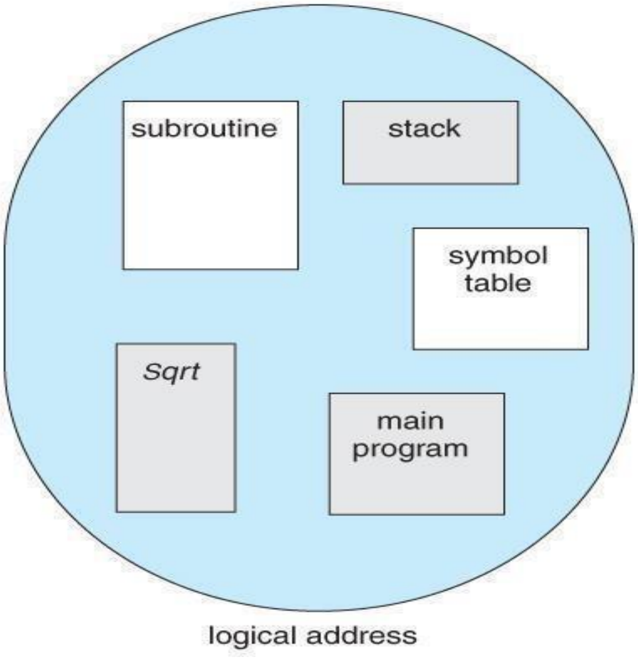


Figure 4.7 Programmer's view of a program.

4.4.2 Segmentation Hardware

- A ***segment table*** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers discussed previously. (Note that at this point in the discussion of segmentation, each segment is kept in contiguous memory and may be of different sizes, but that segmentation can also be combined with paging as we shall see shortly.)

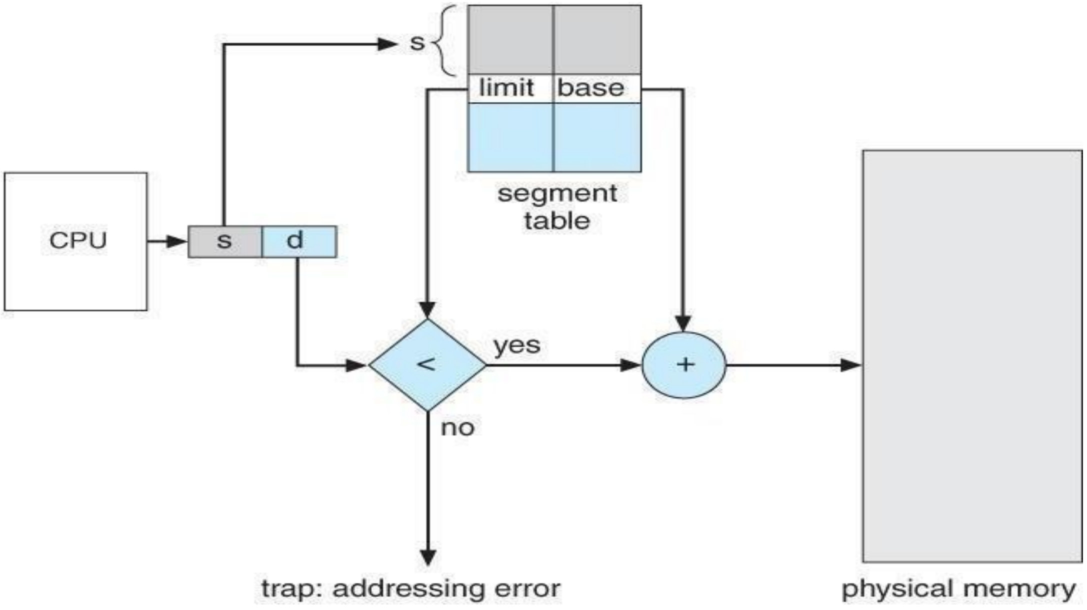


Figure 4.8 - Segmentation hardware

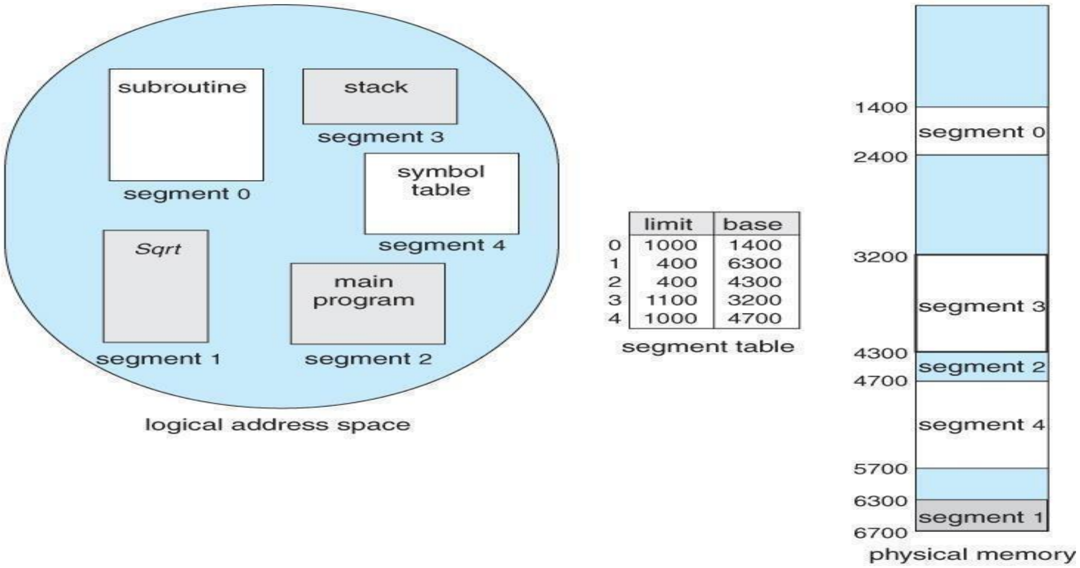


Figure 4.9 - Example of segmentation

4.5 Paging

- Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in equal sized blocks known as ***pages***.
- Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.

4.5.1 Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a programs logical memory space into blocks of the same size called ***pages.***
- Any page (from any process) can be placed into any available frame.
- The ***page table*** is used to look up what frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:

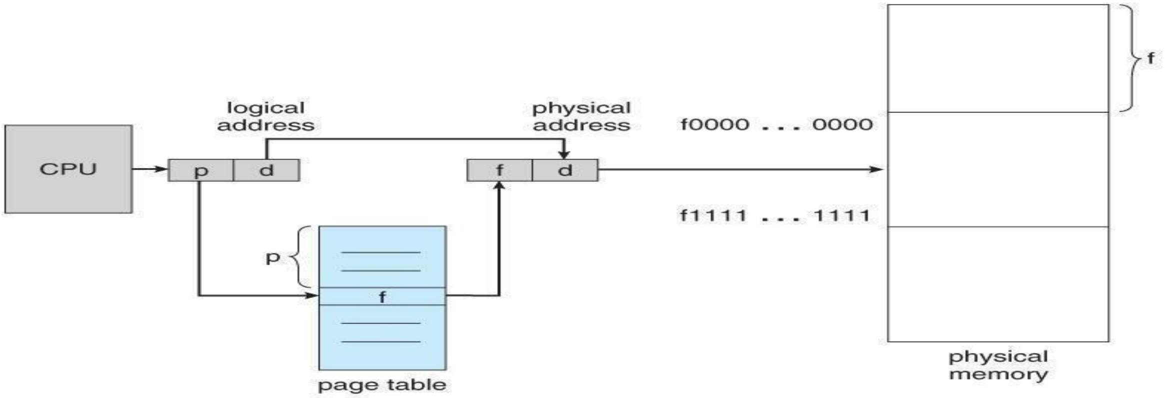


Figure 4.10 - Paging hardware

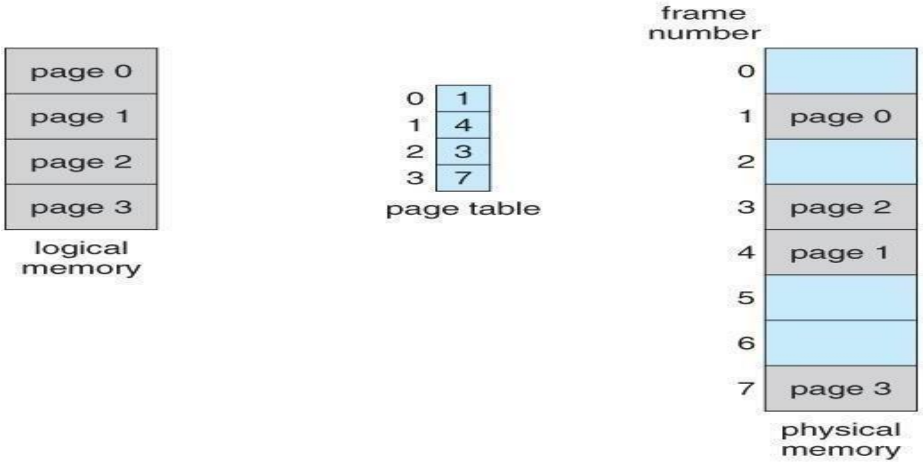
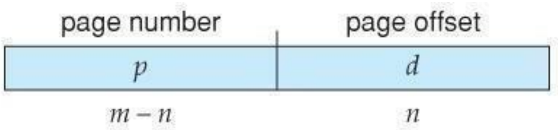


Figure 4.11 - Paging model of logical and physical memory

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

- Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space.



- (DOS used to use an addressing scheme with 16 bit frame numbers and 16-bit offsets, on hardware that only supported 24-bit hardware addresses. The result was a resolution of starting frame addresses finer than the size of a single frame, and multiple frame-offset combinations that mapped to the same physical hardware address.)
- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)

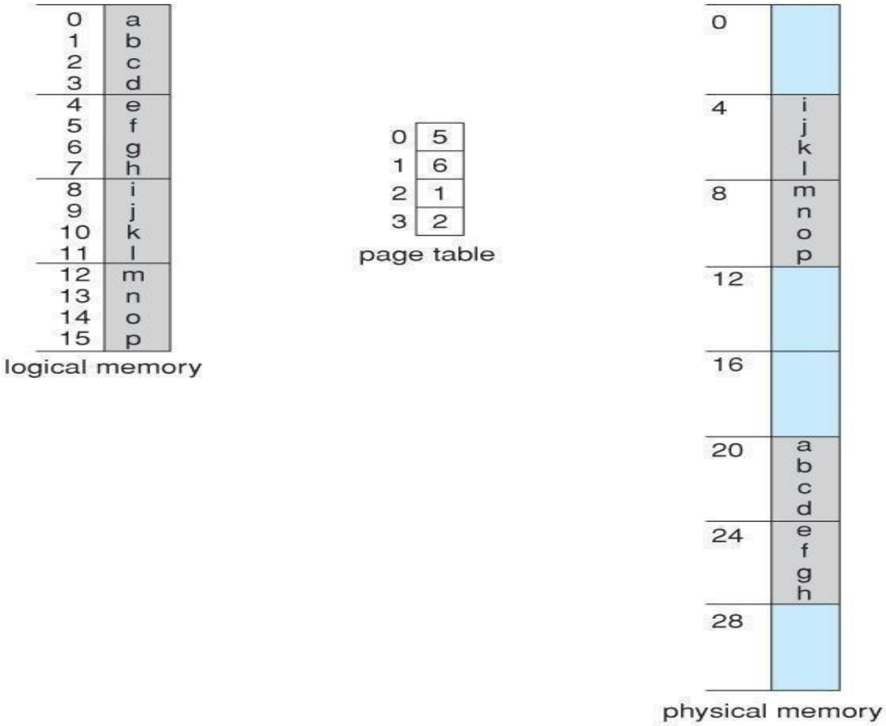


Figure 4.12 - Paging example for a 32-byte memory with 4-byte pages

- Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- There is no external fragmentation with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process. (Possibly more, if processes keep their code and data in separate pages.)

- Larger page sizes waste more memory, but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.
- Page table entries (frame numbers) are typically 32 bit numbers, allowing access to 2^32 physical page frames. If those frames are 4 KB in size each, that translates to 16 TB of addressable physical memory. (32 + 12 = 44 bits of physical address space.)
- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.
- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)

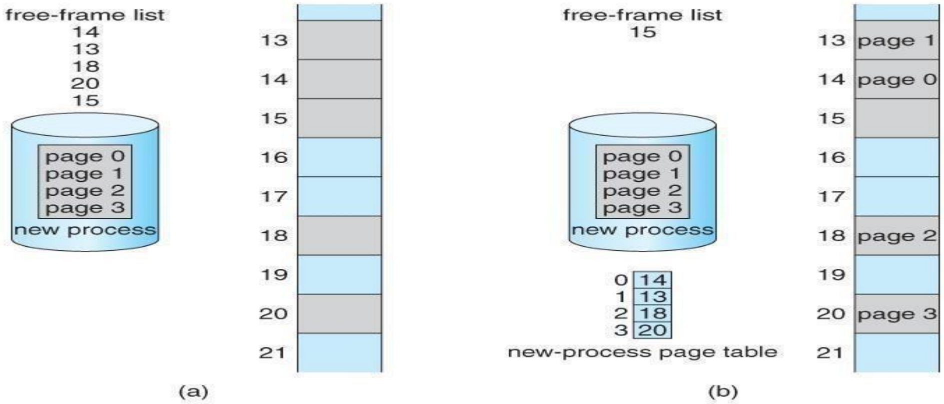


Figure 4.13 - Free frames (a) before allocation and (b) after allocation

4.5.2 Hardware Support

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.
- One option is to use a set of registers for the page table. For example, the DEC PDP-11 uses 16-bit addressing and 8 KB pages, resulting in only 8 pages per process. (It takes 13 bits to address 8 KB of offset, leaving only 3 bits to define a page number.)
- An alternate option is to store the page table in main memory, and to use a single register (called the ***page-table base register, PTBR***) to record where in memory the page table is located.
  - Process switching is fast, because only the single register needs to be changed.
  - However memory access just got half as fast, because every memory access now requires ***two*** memory accesses - One to fetch the frame number from memory and then another one to access the desired memory location.
  - The solution to this problem is to use a very special high-speed memory device called the ***translation look-aside buffer, TLB.***

- The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.

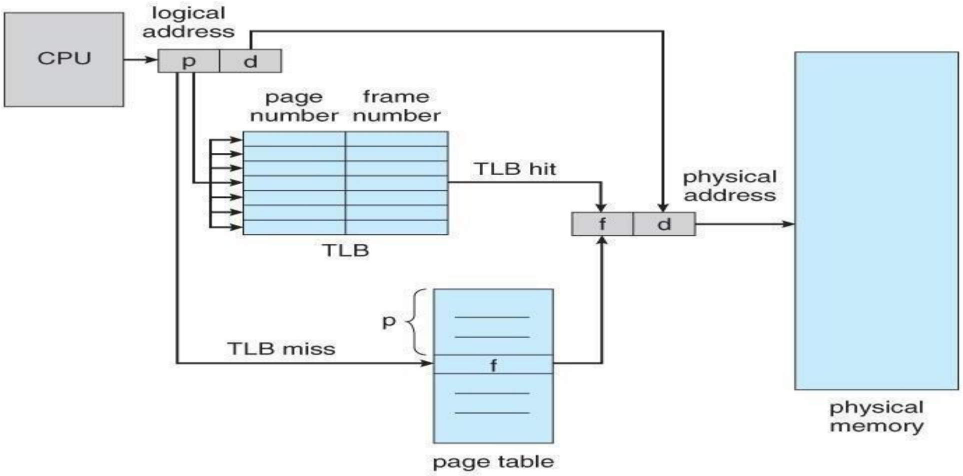


Figure 4.14 - Paging hardware with TLB

The TLB is very expensive, however, and therefore very small. (Not large enough to hold the entire page table.) It is therefore used as a cache device.

Addresses are first checked against the TLB, and if the info is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.

If the TLB is full, then replacement strategies range from ***least-recently used, LRU*** to random.

Some TLBs allow some entries to be ***wired down***, which means that they cannot be removed from the TLB. Typically these would be kernel frames.

Some TLBs store ***address-space identifiers, ASIDs***, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch.

The percentage oftime that the desired information is found in the TLB is termed the ***hit ratio***.

(Eighth Edition Version:) For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be:

0.80 * 120 + 0.20 * 220 = 140 nanoseconds

for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

(Ninth Edition Version:) The ninth edition ignores the 20 nanoseconds required to search the TLB, yielding

0.80 * 100 + 0.20 * 200 = 120 nanoseconds

for a 20% slowdown to get the frame number. A 99% hit rate would yield 101 nanoseconds average access time (you should verify this), for a 1% slowdown.

4.5.3 Protection

The page table can also help to protect processes from accessing memory that they shouldn't, or their own memory in ways that they shouldn't.

A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or some combination of these sorts of things. Then each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.

Valid / invalid bits can be added to "mask off" entries in the page table that are not in use by the current process, as shown by example in Figure 3.12 below.

Note that the valid / invalid bits described above cannot block all illegal memory accesses, due to the internal fragmentation. (Areas of memory in the last page that are not entirely filled by the process, and may contain data left over by whoever used that frame last.)

Many processes do not use all of the page table available to them, particularly in modern systems with very large potential page tables. Rather than waste memory by creating a full-size page table for every process, some systems use a ***page-table length register, PTLR***, to specify the length of the page table.

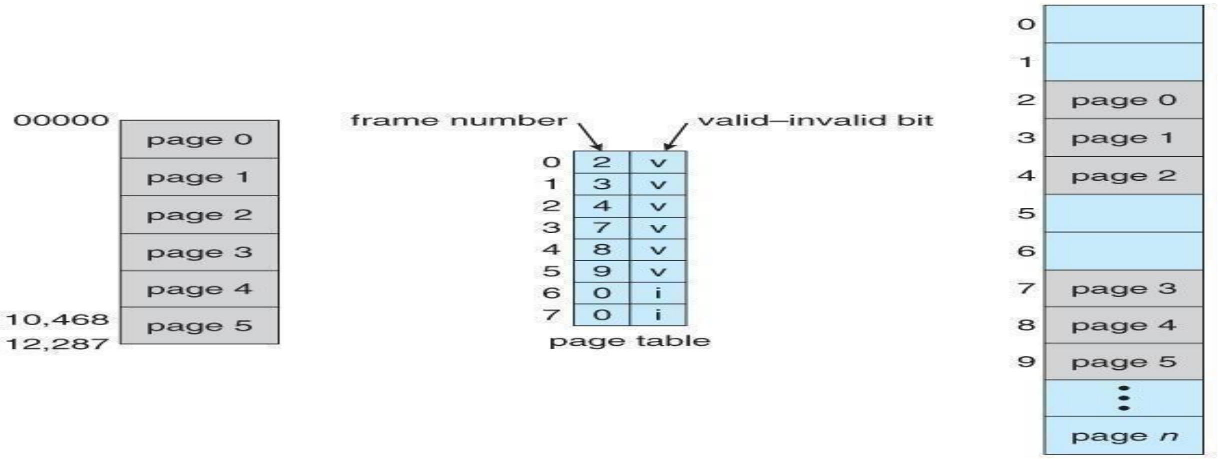


Figure 4.15 - Valid (v) or invalid (i) bit in page table

4.5.4 Shared Pages

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames. This may be done with either code or data.
- If code is ***reentrant***, that means that it does not write to or change the code in any way (it is non self-modifying), and it is therefore safe to re-enter it. More importantly, it means the code can be shared by multiple processes, so long as each has their own copy of the data and registers, including the instruction register.
- In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time.
- Some systems also implement shared memory in this fashion.

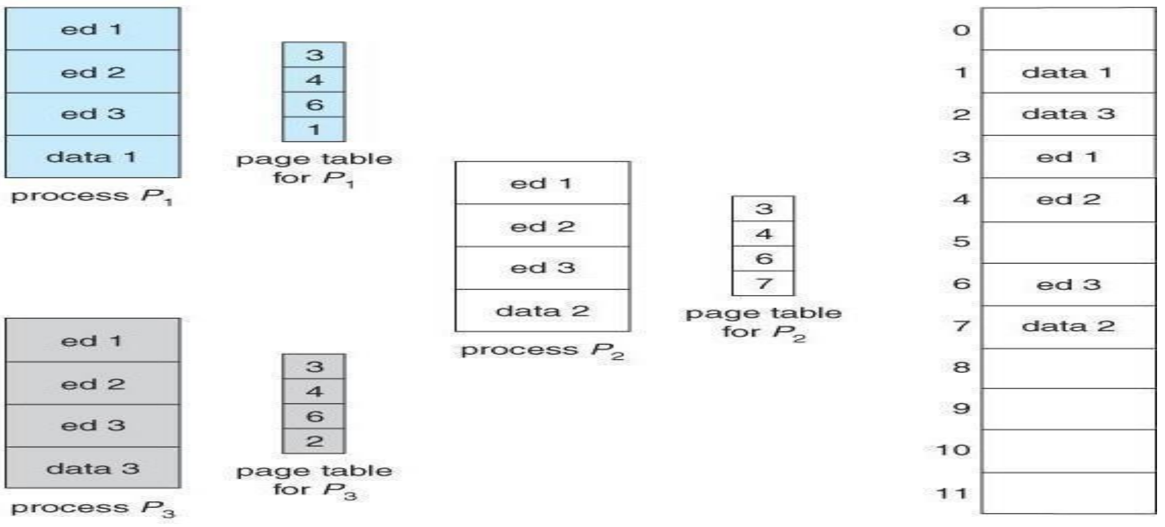
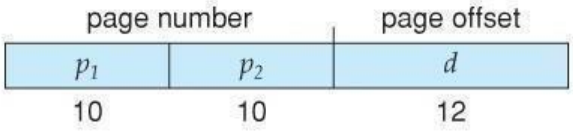


Figure 4.16 - Sharing of code in a paging environment

4.6 Structure of the Page Table

4.6.1 Hierarchical Paging

- Most modern computer systems support logical address spaces of 2^32 to 2^64.
- With a 2^32 address space and 4K (2^12) page sizes, this leave 2^20 entries in the page table. At 4 bytes per entry, this amounts to a 4 MB page table, which is too large to reasonably keep in contiguous memory. (And to swap in and out of memory with each process switch.) Note that with 4K pages, this would take 1024 pages just to hold the page table!
- One option is to use a two-tier paging system, i.e. to page the page table.
- For example, the 20 bits described above could be broken down into two 10-bit page numbers. The first identifies an entry in the outer page table, which identifies where in memory to find one page of an inner page table. The second 10 bits finds a specific entry in that inner page table, which in turn identifies a particular frame in physical memory. (The remaining 12 bits of the 32 bit logical address are the offset within the 4K frame.)



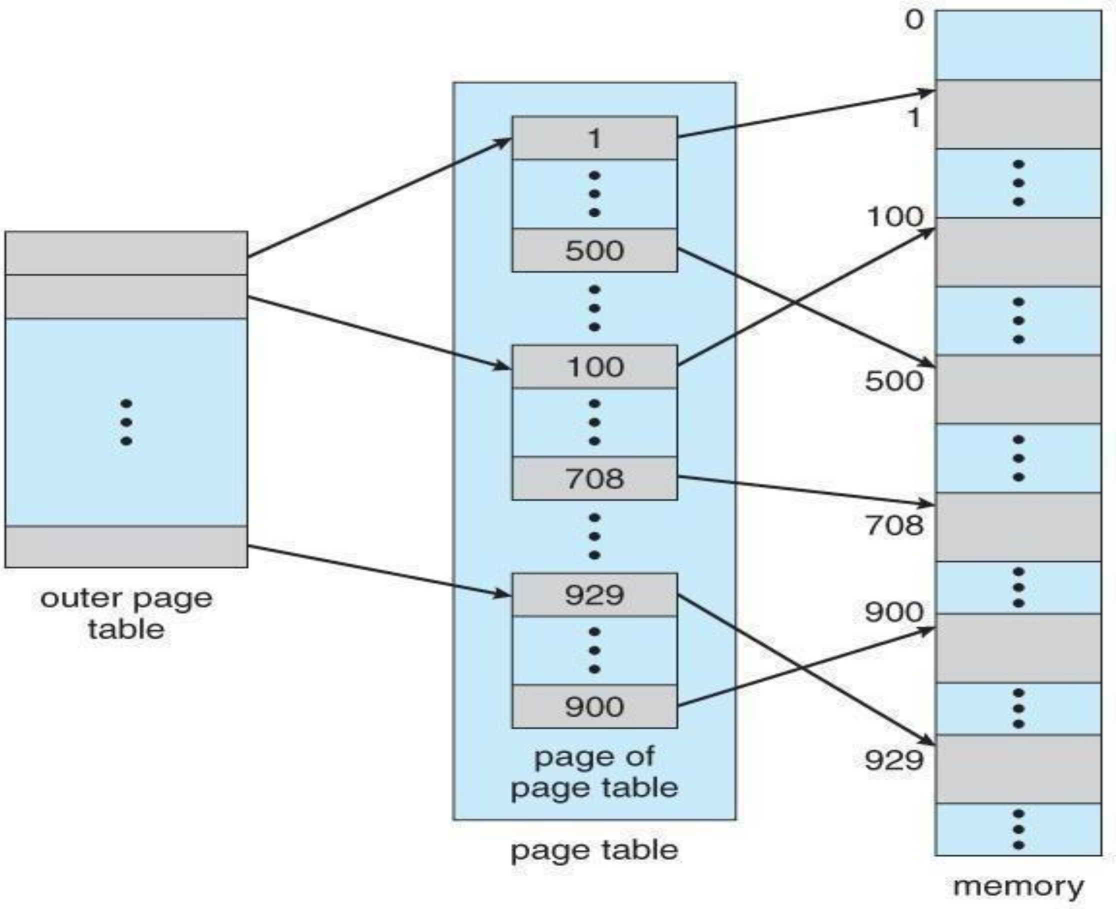


Figure 4.17 A two-level page-table scheme

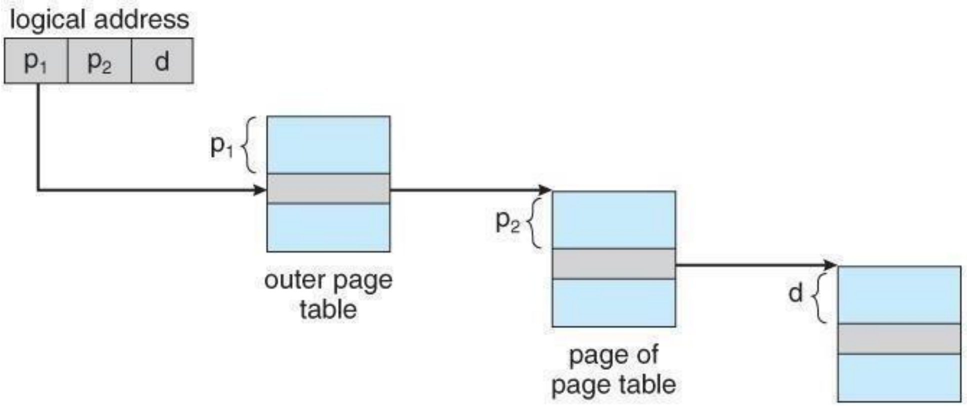
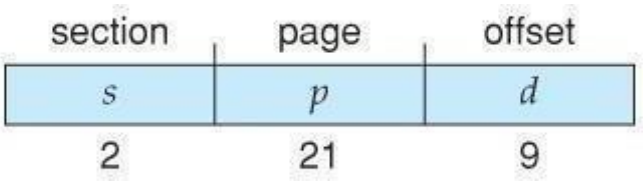
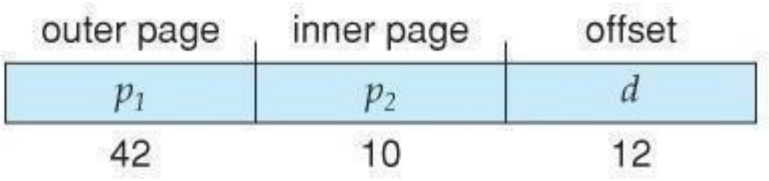


Figure 4.18 - Address translation for a two-level 32-bit paging architecture

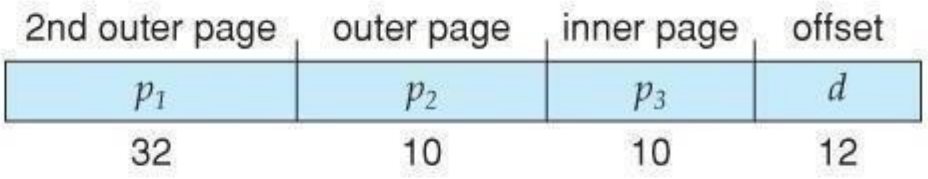
- VAX Architecture divides 32-bit addresses into 4 equal sized sections, and each page is 512 bytes, yielding an address form of:



- With a 64-bit logical address space and 4K pages, there are 52 bits worth of page numbers, which is still too many even for two-level paging. One could increase the paging level, but with 10-bit page tables it would take 7 levels of indirection, which would be prohibitively slow memory access. So some other approach must be used.



64-bits Two-tiered leaves 42 bits in outer table



Going to a fourth level still leaves 32 bits in the outer table.

4.6.2 Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with ***hash tables***. Figure 3.16 below illustrates a ***hashed page table*** using chain-and- bucket hashing:

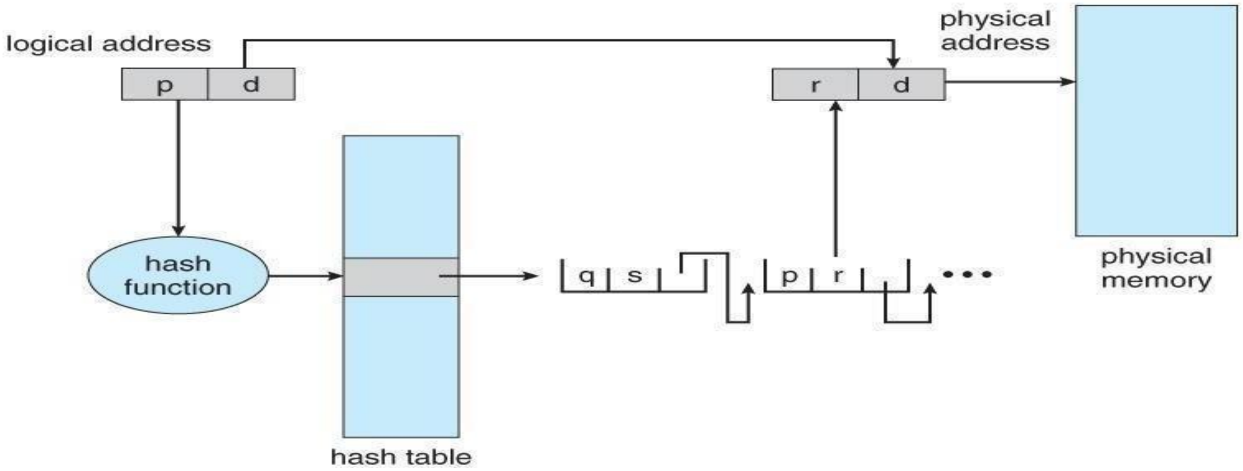


Figure 3.19 - Hashed page table

4.6.3 Inverted Page Tables

- Another approach is to use an ***inverted page table***. Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (I.e. there is one entry per ***frame*** instead of one entry per ***page***.)
- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page (or to discover that it is not there.) Hashing the table can help speedup the search process.
- Inverted page tables prohibit the normal method of implementing shared memory, which is to map multiple logical pages to a common physical frame. (Because each frame is now mapped to one and only one process.)

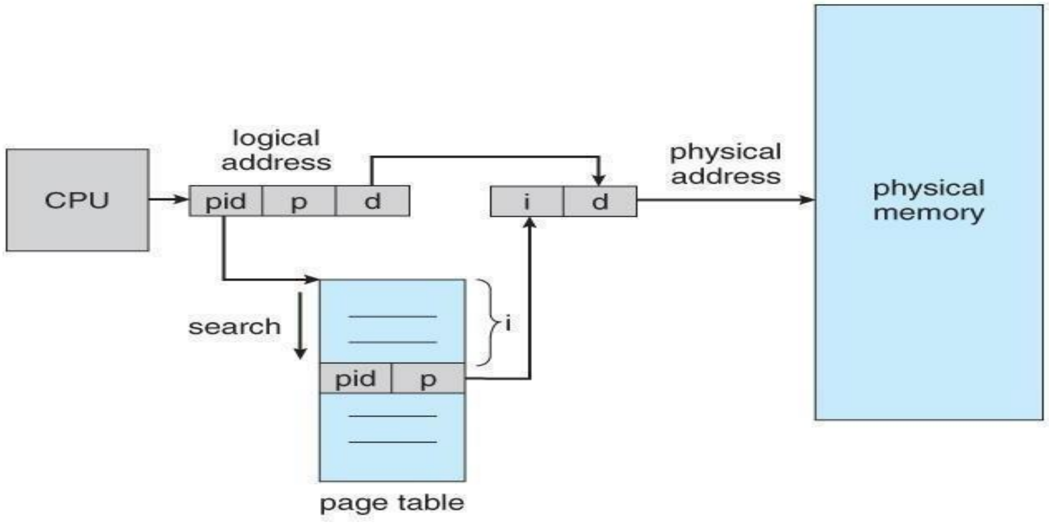


Figure 4.20 - Inverted page table

4.7.1.1 IA-32 Segmentation

- The Pentium CPU provides both pure segmentation and segmentation with paging. In the latter case, the CPU generates a logical address (segment-offset pair), which the segmentation unit converts into a logical linear address, which in turn is mapped to a physical frame by the paging unit, as shown in Figure 3.21:

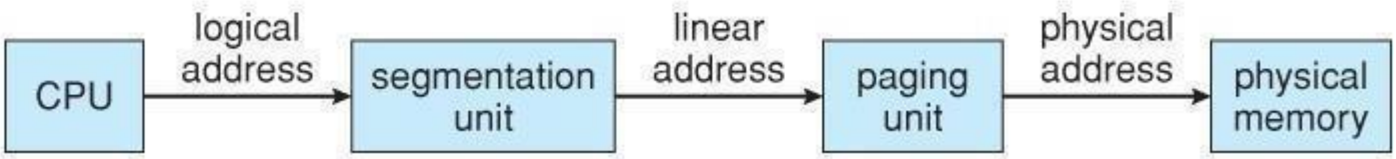
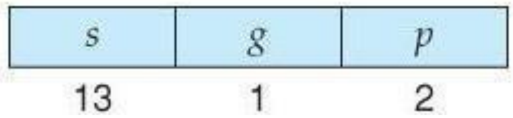


Figure 4.21 - Logical to physical address translation in IA-32

4.7.1.1 IA-32 Segmentation

- The Pentium architecture allows segments to be as large as 4 GB, (24 bits of offset).
- Processes can have as many as 16K segments, divided into two 8K groups:
  - 8K private to that particular process, stored in the ***Local Descriptor Table, LDT.***
  - 8K shared among all processes, stored in the ***Global Descriptor Table, GDT.***
- Logical addresses are (selector, offset) pairs, where the selector is made up of 16 bits:
  - A 13 bit segment number (up to 8K)
  - A 1 bit flag for LDT vs. GDT.
  - 2 bits for protection codes.



- The descriptor tables contain 8-byte descriptions of each segment, including base and limit registers.
- Logical linear addresses are generated by looking the selector up in the descriptor table and adding the appropriate base address to the offset, as shown in Figure 3.22:

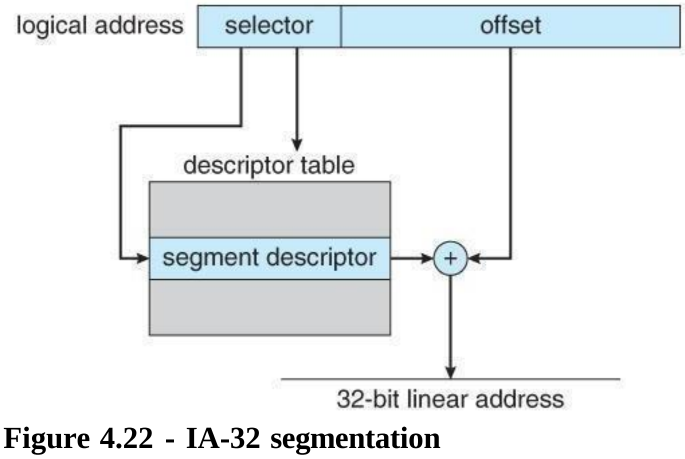
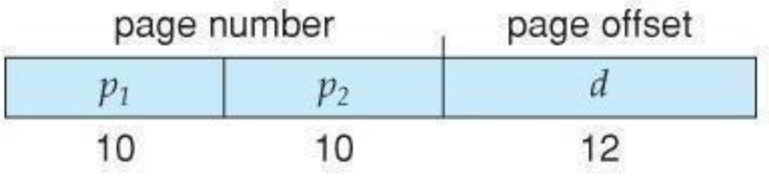


Figure 4.22 - IA-32 segmentation

4.7.1.2 IA-32 Paging

- Pentium paging normally uses a two-tier paging scheme, with the first 10 bits being a page number for an outer page table (a.k.a. page directory), and the next 10 bits being a page number within one of the 1024 inner page tables, leaving the remaining 12 bits as an offset into a 4K page.



- A special bit in the page directory can indicate that this page is a 4MB page, in which case the remaining 22 bits are all used as offset and the inner tier of page tables is not used.
- The CR3 register points to the page directory for the current process, as shown in Figure 8.23 below.
- If the inner page table is currently swapped out to disk, then the page directory will have an "invalid bit" set, and the remaining 31 bits provide information on where to find the swapped out page table on the disk.

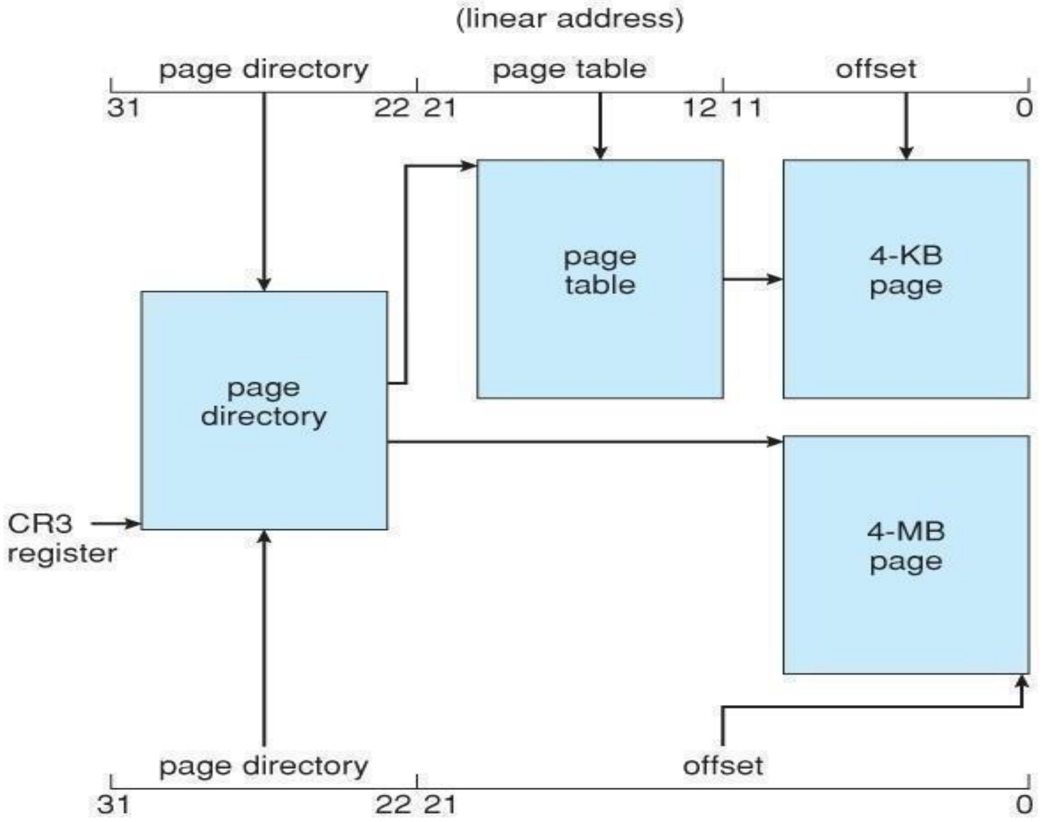


Figure 4.23 - Paging in the IA-32 architecture.

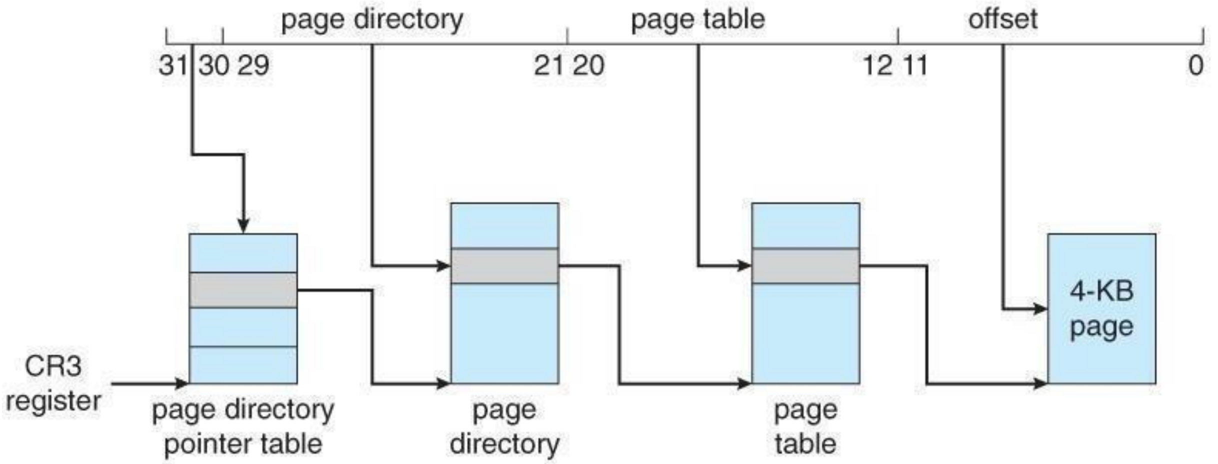


Figure 4.24 - Page address extensions.

VIRTUAL MEMORY

- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:

1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. :-)

- The ability to load only the portions of processes that were actually needed (and only *when* they were needed) has several benefits:

○ Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.

○ Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.

○ Less I/O is needed for swapping processes in and out of RAM, speeding things up.

Figure below shows the general layout of ***virtual memory***, which can be much larger than physical memory:

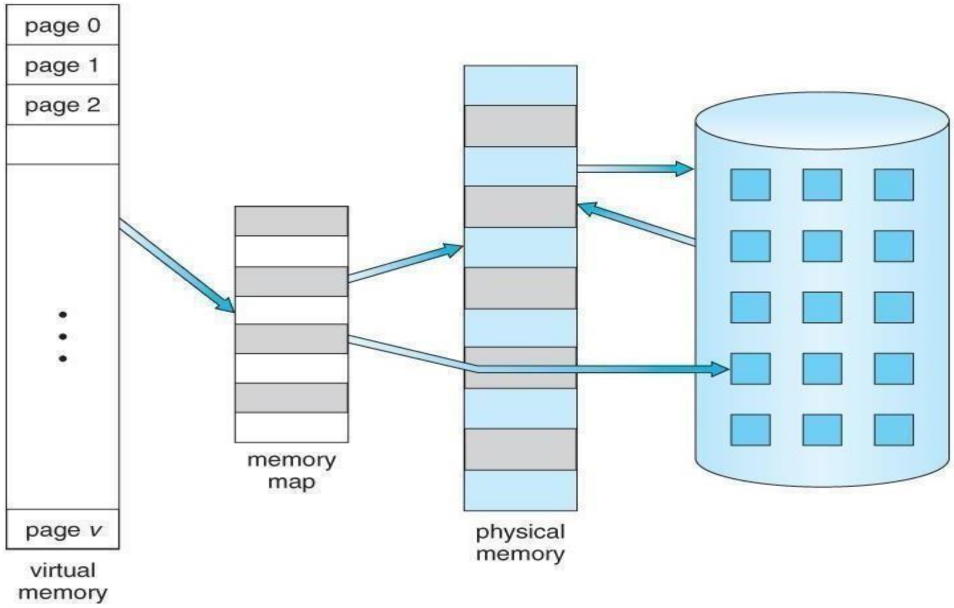


Figure 4.25 - Diagram showing virtual memory that is larger than physical memory

- Figure 4.25 shows ***virtual address space***, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.
- Note that the address space shown in Figure 9.2 is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

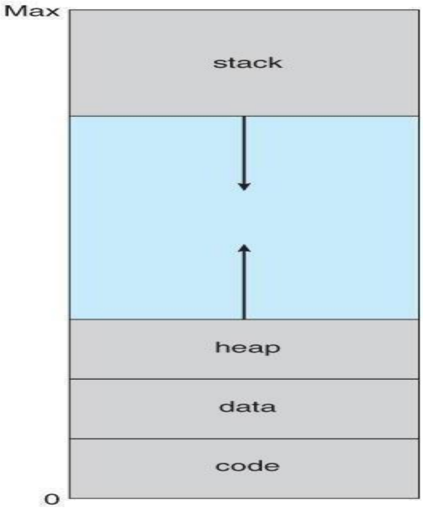


Figure 4.26 - Virtual address space

- Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:
  - System libraries can be shared by mapping them into the virtual address space of more than one process.
  - Processes can also share virtual memory by mapping the same block of memory to more than one process.
  - Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

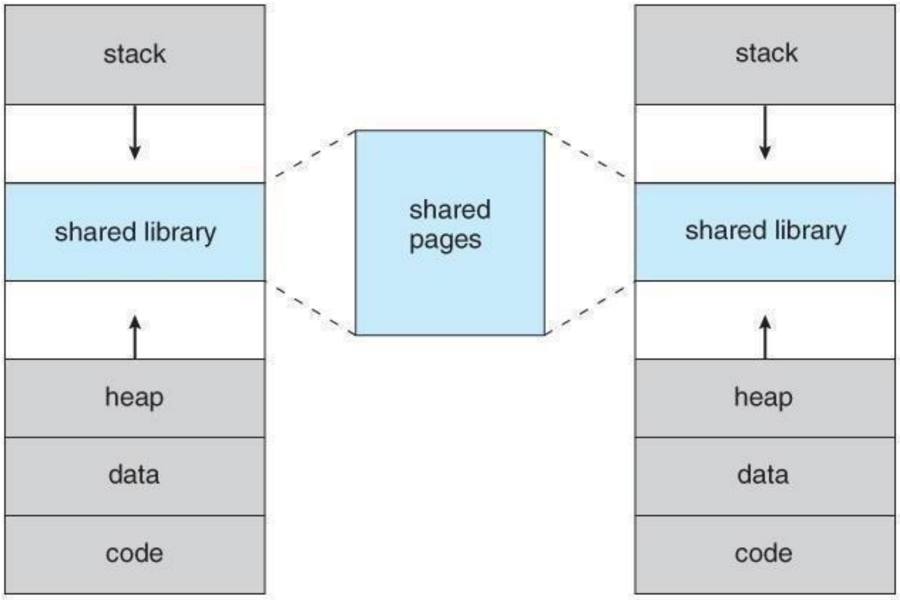


Figure 4.27 - Shared library using virtual memory

4.2 Demand Paging

- The basic idea behind ***demand paging*** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a ***lazy swapper***, although a ***pager*** is a more accurate term.

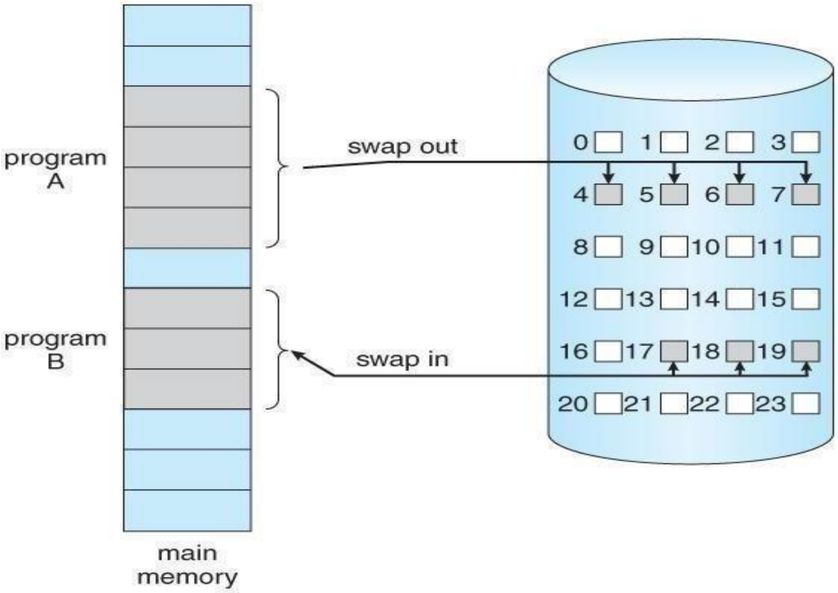


Figure 4.28 - Transfer of a paged memory to contiguous disk space

4.2.1 Basic Concepts

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)

- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)
- If the process only ever accesses pages that are loaded in memory (***memory resident*** pages), then the process runs exactly as if all the pages were loaded in to memory.

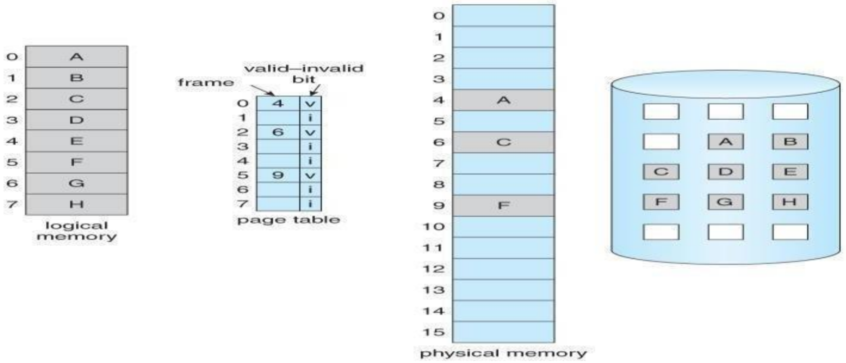


Figure 4.29 - Page table when some pages are not in main memory.

- On the other hand, if a page is needed that was not originally loaded up, then a ***page fault trap*** is generated, which must be handled in a series of steps:

1. The memory address requested is first checked, to make sure it was a valid memory request.
2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)

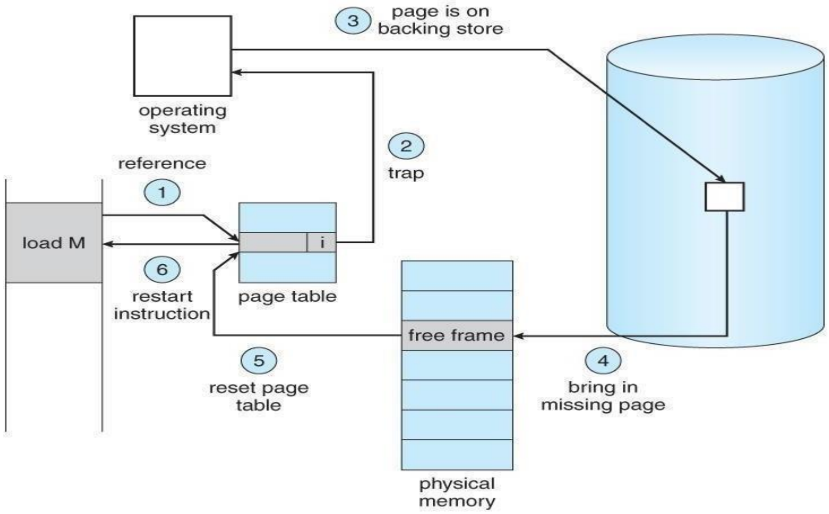


Figure 4.30 - Steps in handling a page fault

- In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as ***pure demand paging.***

- In theory each instruction could generate multiple page faults. In practice this is very rare, due to ***locality of reference***, covered in section 9.6.1.
- The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory. (***Swap space,*** whose allocation is discussed in chapter 12.)
- A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory. For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, (which may span a page boundary), and if some of the data gets modified before the page fault occurs, this could cause problems. One solution is to access both ends of the block before executing the instruction, guaranteeing that the necessary pages get paged in before the instruction begins.

4.2.2 Performance of Demand Paging

- Obviously there is some slowdown and performance hit whenever a page fault occurs and the system has to go get it from memory, but just how big a hit is it exactly?
- There are many steps that occur when servicing a page fault (see book for full details), and some of the steps are optional or variable. But just for the sake of discussion, suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds. (8,000,000 nanoseconds, or 40,000 times a normal memory access.) With a ***page fault rate*** of p, (on a scale from 0 to 1), the effective access time is now:

$$(1 - p) * (200) + p * 8000000 = 200 + 7{,}999{,}800 * p$$

which ***clearly*** depends heavily on p! Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

- A subtlety is that swap space is faster to access than the regular file system, because it does not have to go through the whole directory structure. For this reason some systems will transfer an entire process from the file system to swap space before starting up the process, so that future paging all occurs from the (relatively) faster swap space.
- Some systems use demand paging directly from the file system for binary code (which never changes and hence does not have to be stored on a page operation), and to reserve the swap space for data segments that must be stored. This approach is used by both Solaris and BSD Unix.

4.3 Page Replacement

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process.
- However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider:

1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes. The decision of how to allocate memory for I/O versus user processes is a complex one, yielding different policies on different systems. (Some allocate a fixed amount for I/O, and others let the I/O system contend for memory along with everything else.)
2. Put the process requesting more pages into a wait queue until some free frames become available.

3. Swap some process out of memory completely, freeing up its page frames.
4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as ***page replacement***, and is the most common solution. There are many different algorithms for page replacement, which is the subject of the remainder of this section.

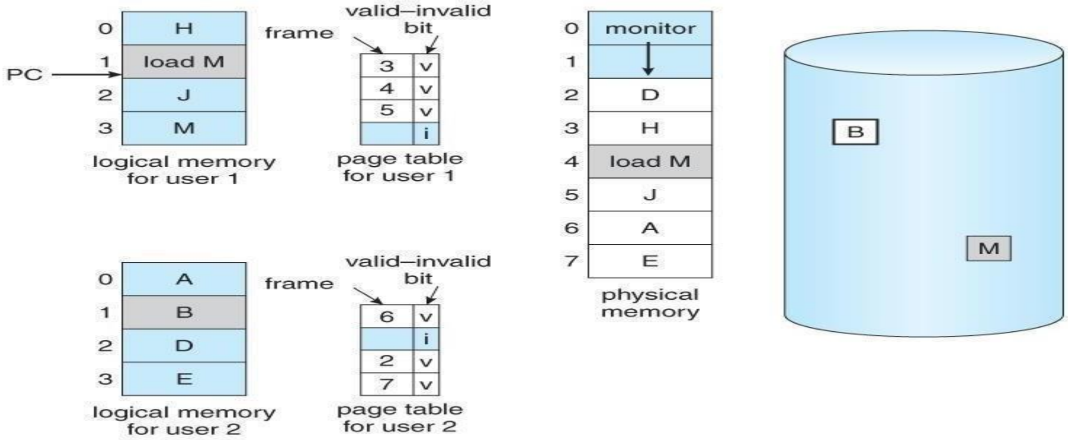


Figure 4.31 - Need for page replacement.

4.3.1 Basic Page Replacement

- The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:

1. Find the location of the desired page on the disk, either in swap space or in the file system.
2. Find a free frame:

a. If there is a free frame, use it.
b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the ***victim frame***.
c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.

3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
4. Restart the process that was waiting for this page.

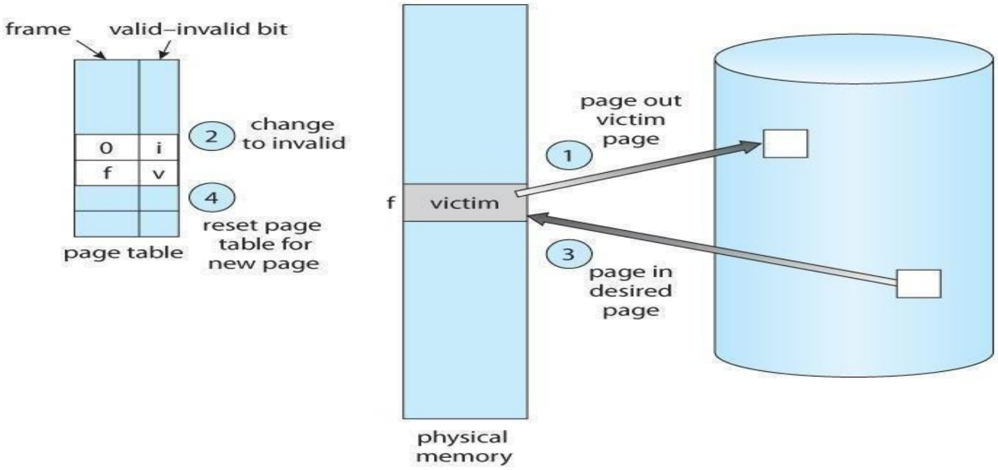


Figure 4.32 - Page replacement.

- Note that step 3c adds an extra disk write to the page-fault handling, effectively doubling the time required to process a page fault. This can be alleviated somewhat by assigning a ***modify bit,*** or ***dirty bit*** to each page, indicating whether or not it has been changed since it was last loaded in from disk. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required. It should come as no surprise that many page replacement strategies specifically look for pages that do not have their dirty bit set, and preferentially select clean pages as victim pages. It should also be obvious that unmodifiable code pages never get their dirty bits set.
- There are two major requirements to implement a successful demand paging system. We must develop a ***frame-allocation algorithm*** and a ***page-replacement algorithm.*** The former centers around how many frames are allocated to each process (and to other needs), and the latter deals with how to select a page for replacement when there are no free frames available.
- The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.
- Algorithms are evaluated using a given string of memory accesses known as a ***reference string,*** which can be generated in one of(at least) three common ways:

1. Randomly generated, either evenly distributed or with some distribution curve based on observed system behavior. This is the fastest and easiest approach, but may not reflect real performance well, as it ignores locality of reference.
2. Specifically designed sequences. These are useful for illustrating the properties of comparative algorithms in published papers and textbooks, (and also for homework and exam problems. :-))
3. Recorded memory references from a live system. This may be the best approach, but the amount of data collected can be enormous, on the order of a million addresses per second. The volume of collected data can be reduced by making two important observations:

1. Only the page number that was accessed is relevant. The offset within that page does not affect paging operations.
2. Successive accesses within the same page can be treated as a single page request, because all requests after the first are guaranteed to be page hits. (Since there are no intervening requests for other pages that could remove this page from the page table.)

▪ So for example, if pages were of size 100 bytes, then the sequence of address requests (0100, 0432, 0101, 0612, 0634, 0688, 0132, 0038, 0420) would reduce to page requests (1, 4, 1, 6, 1, 0, 4)

As the number of available frames increases, the number of page faults should decrease, as shown in Figure 3.33:

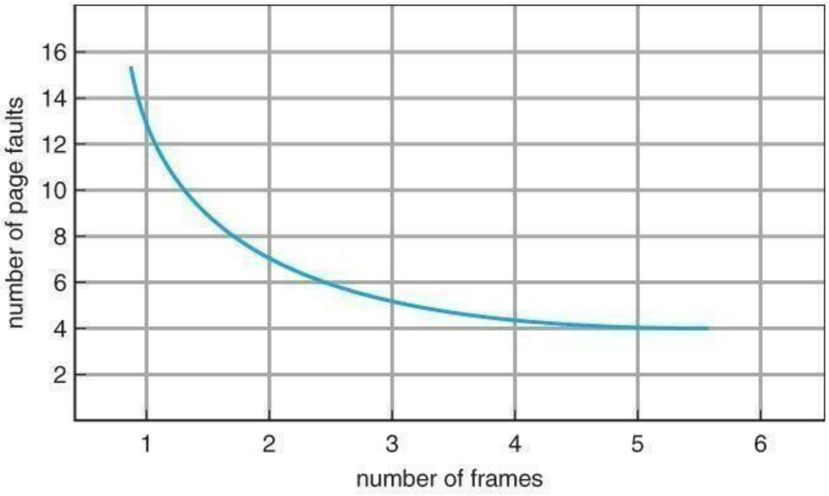


Figure 4.33 - Graph of page faults versus number of frames.

4.3.2 FIFO Page Replacement

- A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults:

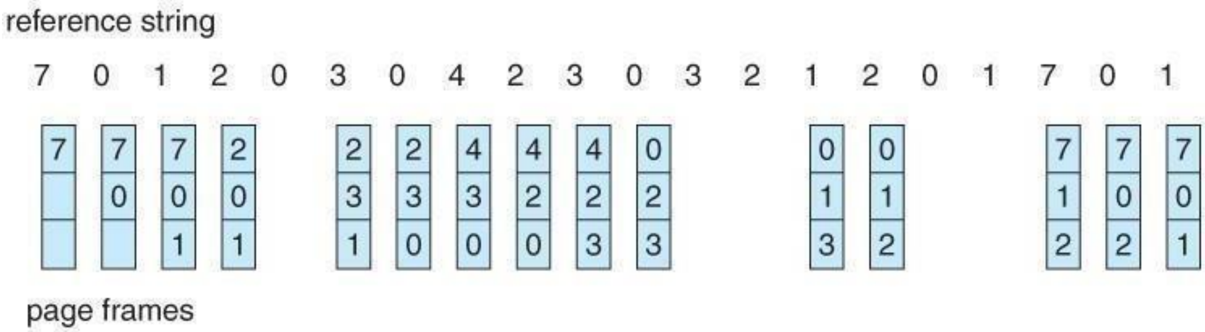


Figure 4 .34 - FIFO page-replacement algorithm.

- Although FIFO is simple and easy, it is not always optimal, or even efficient.
- An interesting effect that can occur with FIFO is ***Belady's anomaly***, in which increasing the number of frames available can actually ***increase*** the number of page faults that occur! Consider, for example, the following chart based on the page sequence (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and a varying number of available frames. Obviously the maximum number of faults is 12 (every request generates a fault), and the minimum number is 5 (each page loaded only once), but in between there are some interesting results:

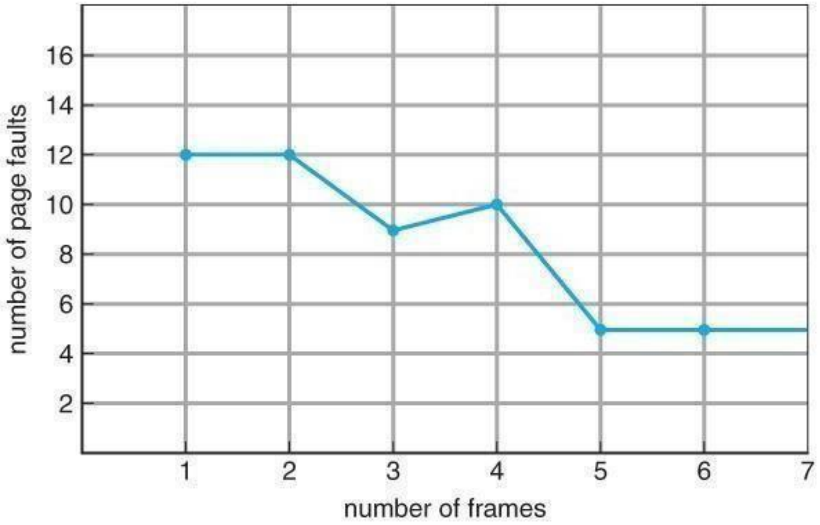


Figure 4.35 - Page-fault curve for FIFO replacement on a reference string.

4.3.3 Optimal Page Replacement

- The discovery of Belady's anomaly lead to the search for an ***optimal page-replacement algorithm***, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called ***OPT orMIN.*** This algorithm is simply "Replace the page that will not be used for the longest time in the future."

- For example, Figure 9.14 shows that by applying OPT to the same reference string used for the FIFO example, the minimum number of possible page faults is 9. Since 6 of the page-faults are unavoidable (the first reference to each new page), FIFO can be shown to require 3 times as many (extra) page faults as the optimal algorithm. (Note: The book claims that only the first three page faults are required by all algorithms, indicating that FIFO is only twice as bad as OPT.)
- Unfortunately OPT cannot be implemented in practice, because it requires foretelling the future, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.
- In practice most page-replacement algorithms try to approximate OPT by predicting (estimating) in one fashion or another what page will not be used for the longest period of time. The basis of FIFO is the prediction that the page that was brought in the longest time ago is the one that will not be needed again for the longest future time, but as we shall see, there are many other prediction methods, all striving to match the performance of OPT.

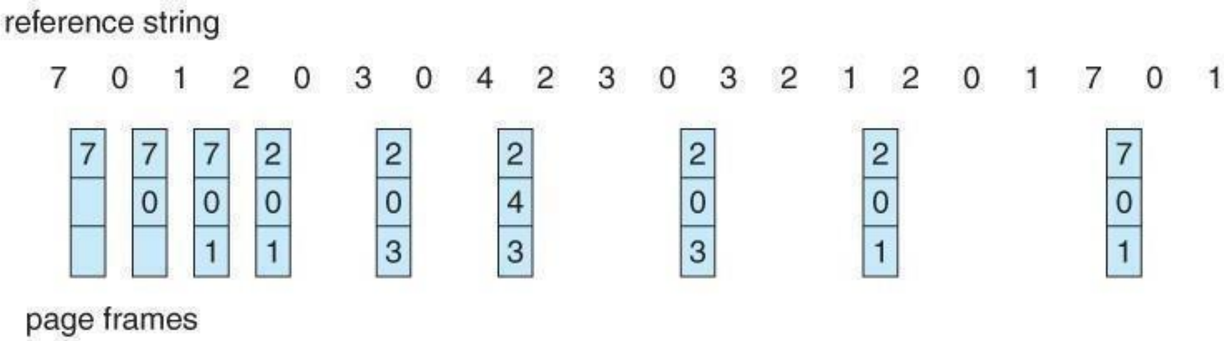


Figure 4.36 - Optimal page-replacement algorithm

4.4 LRU Page Replacement

- The prediction behind ***LRU,*** the ***Least Recently Used,*** algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest ***load*** time, and the latter looks at the oldest ***use*** time.)
- Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.)
- Figure 9.15 illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

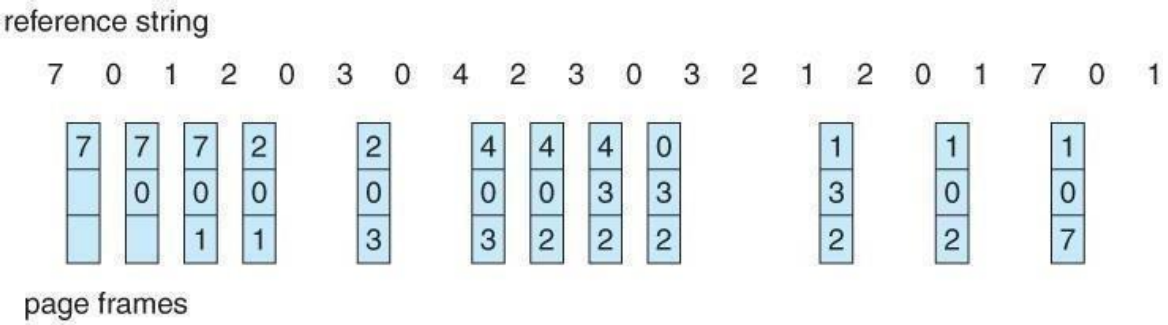


Figure 4.37 - LRU page-replacement algorithm.

- LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used:

1. **Counters.** Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.
2. **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

- Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for ***every*** memory access.
- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called ***stack algorithms,*** which can never exhibit Belady's anomaly. A stack algorithm is one in which the pages kept in memory for a frame set of size N will always be a subset of the pages kept for a frame size of N + 1. In the case of LRU, (and particularly the stack implementation thereof), the top N pages of the stack will be the same for all frame set sizes of N or anything larger.

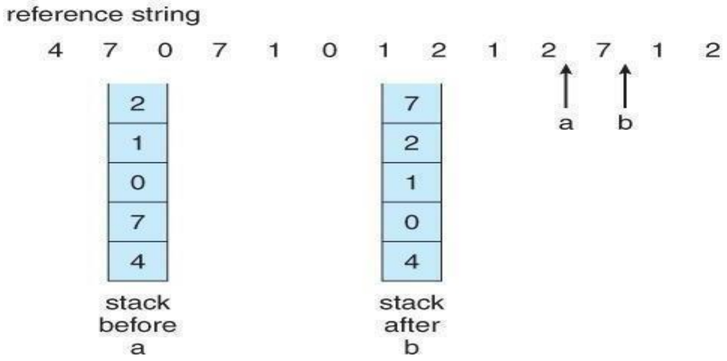


Figure 4.38 - Use of a stack to record the most recent page references.

4.5 LRU-Approximation Page Replacement

- Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary.
- However many systems offer some degree of HW support, enough to approximate LRU fairly well. (In the absence of ANY hardware support, FIFO might be the best available choice.)
- In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to distinguish pages that have been accessed since the last clear from those that have not, but does not provide any finer grain of detail.

4.5.1 Additional-Reference-Bits Algorithm

- Finer grain is possible by storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry, which is interpreted as an unsigned int.
  - At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit.
  - The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared.

- At any given time, the page with the smallest value for the reference byte is the LRU page.
- Obviously the specific number of bits used and the frequency with which the reference byte is updated are adjustable, and are tuned to give the fastest performance on a given hardware platform.

4.5.2 Second-Chance Algorithm

- The ***second chance algorithm*** is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table.
- When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
- If a page is found with its reference bit not set, then that page is selected as the next victim.
- If, however, the next page in the FIFO **does** have its reference bit set, then it is given a second chance:
- The reference bit is cleared, and the FIFO search continues.
- If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page table.
- If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.
- If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement.
- As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely.
- This algorithm is also known as the ***clock*** algorithm, from the hands of the clock moving around the circular queue.

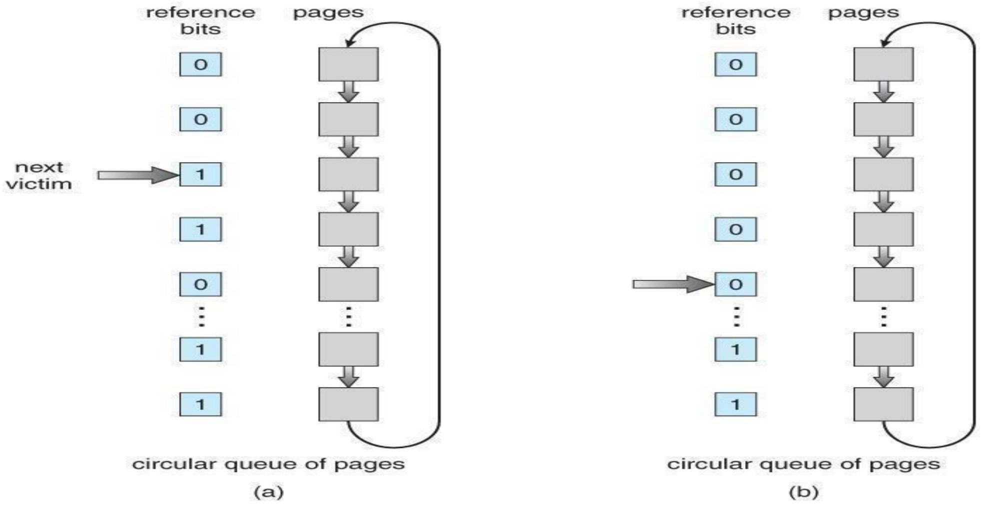


Figure 4.39 - Second-chance (clock) page-replacement algorithm.

4.5.3 Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:

1.(0, 0) - Neither recently used nor modified.
2.(0, 1) - Not recently used, but modified.
3.(1, 0) - Recently used, but clean.
4.(1, 1) - Recently used and modified.

- This algorithm searches the page table in a circular fashion (in as many as four passes), looking for the first page it can find in the lowest numbered category. I.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

4.6 Counting-Based Page Replacement

- There are several algorithms based on counting the number of references that have been made to a given page, such as:
  - ***Least Frequently Used, LFU:*** Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to right-shift the counters periodically, yielding a time-decaying average reference count.
  - ***Most Frequently Used, MFU:*** Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.
- In general counting-based algorithms are not commonly used, as their implementation is expensive and they do not approximate OPT well.

4.7 Page-Buffering Algorithms

There are a number of page-buffering algorithms that can be used in conjunction with the afore- mentioned algorithms, to improve overall performance and sometimes make up for inherent weaknesses in the hardware and/or the underlying page-replacement algorithms:

- Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, to get the requesting process up and running again as quickly as possible, and then select a victim page to write to disk and free up a frame as a second step.
- Keep a list of modified pages, and when the I/O system is otherwise idle, have it write these pages out to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim.
- Keep a pool of free frames, but remember what page was in it before it was made free. Since the data in the page is not actually cleared out when the page is freed, it can be made an active page again without having to load in any new data from disk. This is useful when an algorithm mistakenly replaces a page that in fact is needed again soon.

4.8 Applications and Page Replacement

- Some applications (most notably database programs) understand their data accessing and caching needs better than the general-purpose OS, and should therefore be given reign to do their own memory management.
- Sometimes such programs are given a ***raw disk partition*** to work with, containing raw data blocks and no file system structure. It is then up to the application to use this disk partition as extended memory or for whatever other reasons it sees fit.

4.4 Allocation of Frames

We said earlier that there were two important tasks in virtual memory management: a page-replacement strategy and a frame-allocation strategy. This section covers the second part of that pair.

4.4.1 Minimum Number of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture, and corresponds to the worst-case scenario of the number of pages that could be touched by a single (machine) instruction.
- If an instruction (and its operands) spans a page boundary, then multiple pages could be needed just for the instruction fetch.
- Memory references in an instruction touch more pages, and if those memory locations can span page boundaries, then multiple pages could be needed for operand access also.
- The worst case involves indirect addressing, particularly where multiple levels of indirect addressing are allowed. Left unchecked, a pointer to a pointer to a pointer to a pointer to a . . . could theoretically touch every page in the virtual address space in a single machine instruction, requiring every virtual page be loaded into physical memory simultaneously. For this reason architectures place a limit (say 16) on the number of levels of indirection allowed in an instruction, which is enforced with a counter initialized to the limit and decremented with every level of indirection in an instruction - If the counter reaches zero, then an "excessive indirection" trap occurs. This example would still require a minimum frame allocation of 17 per process.

4.4.2 Allocation Algorithms

- **Equal Allocation -** If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation -** Allocate the frames proportionally to the size of the process, relative to the total size of all processes. So if the size of process i is S_i, and S is the sum of all S_i, then the allocation for process P_i is a_i = m * S_i / S.
- Variations on proportional allocation could consider priority of process rather than just their size.
- Obviously all allocations fluctuate over time as the number of available free frames, m, fluctuates, and all are also subject to the constraints of minimum allocation. (If the minimum allocations cannot be met, then processes must either be swapped out or not allowed to start until more free frames become available.)

4.4.3 Global versus Local Allocation

- One big question is whether frame allocation (page replacement) occurs on a local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient, and is the more commonly used approach.

4.4.4 Non-Uniform Memory Access

- The above arguments all assume that all memory is equivalent, or at least has equivalent access times.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.

- In these latter systems, CPUs can access memory that is physically located on the same board much faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.
- The presence of threads complicates the picture, especially when the threads get loaded onto different processors.
- Solaris uses an ***lgroup*** as a solution, in a hierarchical fashion based on relative latency. For example, all processors and RAM on a single board would probably be in the same lgroup. Memory assignments are made within the same lgroup if possible, or to the next nearest lgroup otherwise. (Where "nearest" is defined as having the lowest access time.)

4.5 Thrashing

- If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling.
- But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.
- A process that is spending more time paging than executing is said to be ***thrashing.***

4.5.1 Cause of Thrashing

- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.
- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the schedule to add in even more processes and exacerbating the problem! Eventually the system would essentially grind to a halt.
- Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (or any other I/O for that matter.)

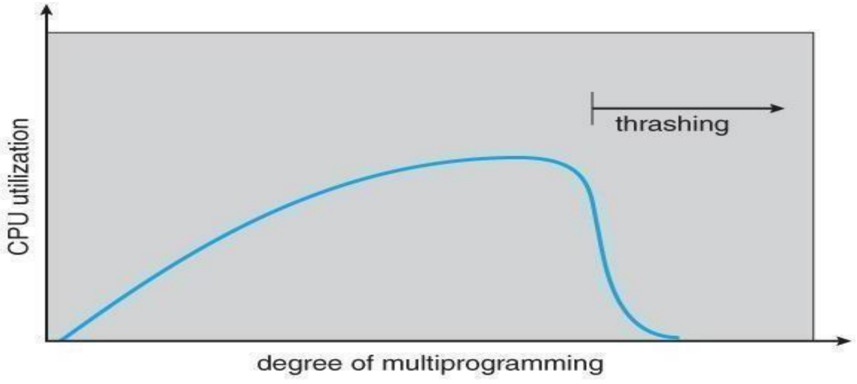


Figure 4.40 - Thrashing

- To prevent thrashing we must provide processes with as many frames as they really need "right now", but how do we know what that is?

4.5.2 Working-Set Model

- The ***working set model*** is based on the concept of locality, and defines a ***working set window***, of length ***delta.*** Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set, as illustrated in Figure 9.20:

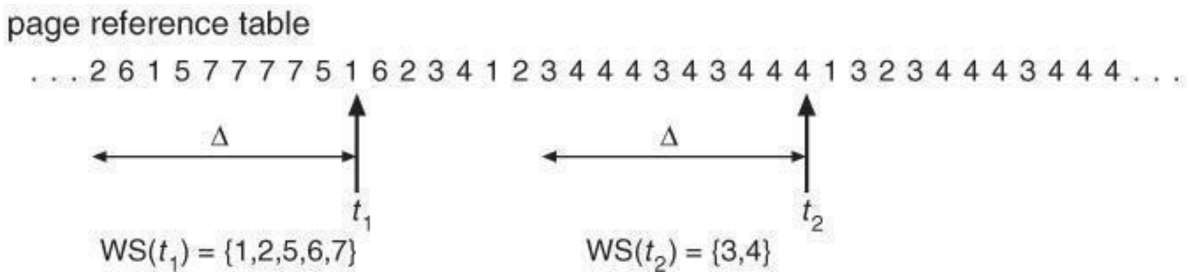


Figure 4.41 - Working-set model.

- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.
- The total demand, D, is the sum of the sizes of the working sets for all processes. If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched.
- The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page. An approximation can be made using reference bits and a timer that goes off after a set interval of memory references:
  - For example, suppose that we set the timer to go off after every 5000 references (by any process), and we can store two additional historical reference bits in addition to the current reference bit.
  - Every time the timer goes off, the current reference bit is copied to one of the two historical bits, and then cleared.
  - If any of the three bits is set, then that page was referenced within the last 15,000 references, and is considered to be in that processes reference set.
  - Finer resolution can be achieved with more historical bits and a more frequent timer, at the expense of greater overhead.

4.5.3 Page-Fault Frequency

- A more direct approach is to recognize that what we really want to control is the page-fault rate, and to allocate frames based on this directly measurable value. If the page-fault rate exceeds a certain upper bound then that process needs more frames, and if it is below a given lower bound, then it can afford to give up some of its frames to other processes.
- (I suppose a page-replacement strategycould be devised that would select victim frames based on the process with the lowest current page-fault frequency.)

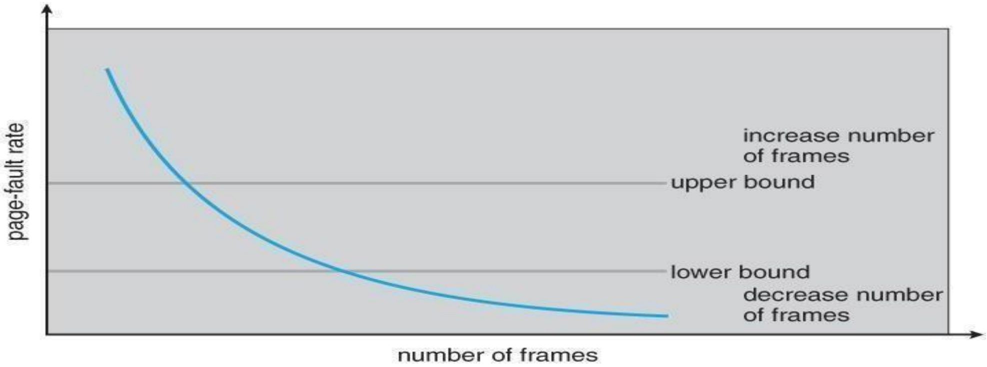


Figure 4.42 - Page-fault frequency.

- Note that there is a direct relationship between the page-fault rate and the working-set, as a process moves from one locality to another:

UNIT- V

File System Interface and Operations -Access methods, Directory Structure, Protection, File System Structure, Allocation methods, kernel support for files, system calls for file I/O operations open, create, read, write, close, lseek, stat, ioctl

Disk Management: Disk Scheduling Algorithms-FCFS, SSTF, SCAN, C-SCAN

FILE-SYSTEM INTERFACE

File Attributes

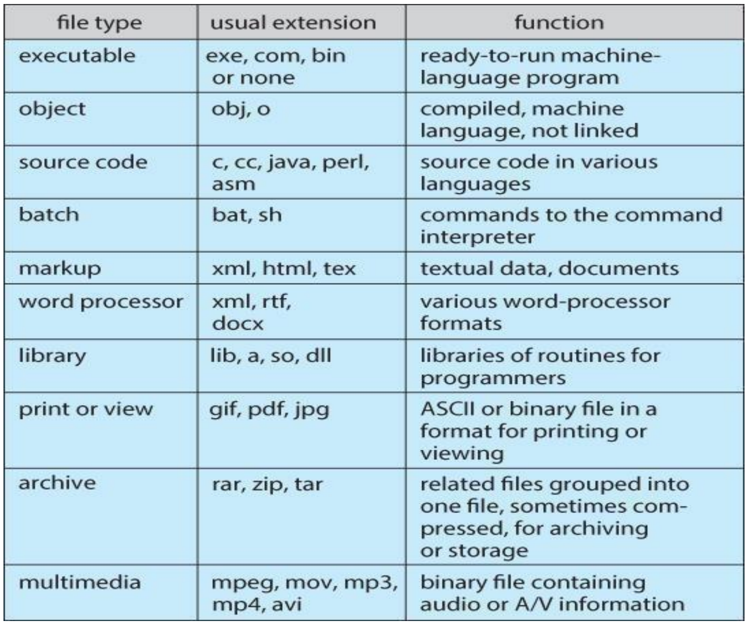
- Different OSes keep track of different file attributes, including:
  - **Name -** Some systems give special significance to names, and particularly extensions (.exe, .txt, etc.), and some do not. Some extensions may be of significance to the OS (.exe), and others only to certain applications (.jpg)
  - **Identifier** (e.g. inode number)
  - **Type -** Text, executable, other binary, etc.
  - **Location -** on the hard drive.
  -
  - **Size**
  - **Protection**
  - **Time & Date**
  - **User ID**

File Operations

- The file ADT supports many common operations:
  - Creating a file
  - Writing a file
  - Reading a file
  - Repositioning within a file
  - Deleting a file
  - Truncating a file.
- Most OSes require that files be ***opened*** before access and ***closed*** after all access is complete. Normally the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an ***open file table***, containing for example:
  - **File pointer -** records the current position in the file, for the next read or write access.
  - **File-open count -** How many times has the current file been opened (simultaneously by different processes) and not yet closed? When this counter reaches zero the file can be removed from the table.
  - **Disk location of the file.**
  - **Access rights**
- Some systems provide support for ***file locking.***
  - A ***shared lock*** is for reading only.
  - A ***exclusive lock*** is for writing as well as reading.
  - An ***advisory lock*** is informational only, and not enforced. (A "Keep Out" sign, which may be ignored.)
  - A ***mandatory lock*** is enforced. (A truly locked door.)
  - UNIX used advisory locks, and Windows uses mandatory locks.

File Types

- Windows (and some other systems) use special file extensions to indicate the type of each file:



file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Common file types.

- Macintosh stores a creator attribute for each file, according to the program that first created it with the create( ) system call.
- UNIX stores magic numbers at the beginning of certain files. (Experiment with the "file" command, especially in directories such as /bin and /dev)

File Structure

- Some files contain an internal structure, which may or may not be known to the OS.
- For the OS to support particular file formats increases the size and complexity ofthe OS.
- UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.)
- Macintosh files have two ***forks*** - a ***resource fork***, and a ***data fork***. The resource fork contains information relating to the UI, such as icons and button images, and can be modified independently of the data fork, which contains the code or data as appropriate.

Internal File Structure

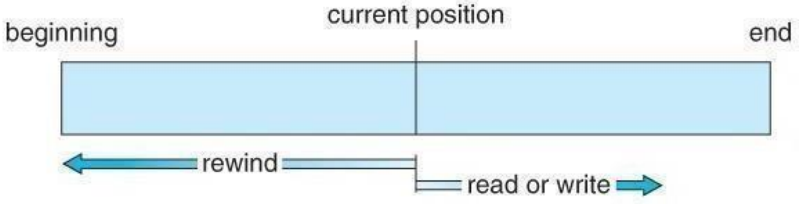
- Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof. (Larger physical disks use larger block sizes, to keep the range of block numbers within the range of a 32-bit integer.)
- Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or structure size.
- The number of logical units which fit into one physical block determines its ***packing***, and has an impact on the amount of internal fragmentation (wasted space) that occurs.

- As a general rule, half a physical block is wasted for each file, and the larger the block sizes the more space is lost to internal fragmentation.

Access Methods

Sequential Access

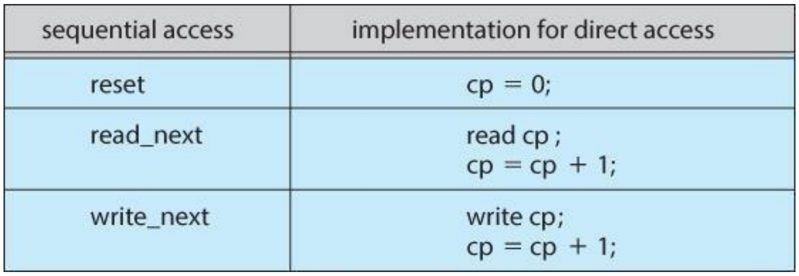
- A sequential access file emulates magnetic tape operation, and generally supports a few operations:
  - read next - read a record and advance the tape to the next position.
  - write next - write a record and advance the tape to the next position.
  - rewind
  - skip n records - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.



Sequential-access file.

Direct Access

- Jump to any record and read that record. Operations supported include:
  - read n - read record number n. (Note an argument is now required.)
  - write n - write record number n. (Note an argument is now required.)
  - jump to record n - could be 0 or the end of file.
  - Query current record - used to return back to this record later.
  - Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient.



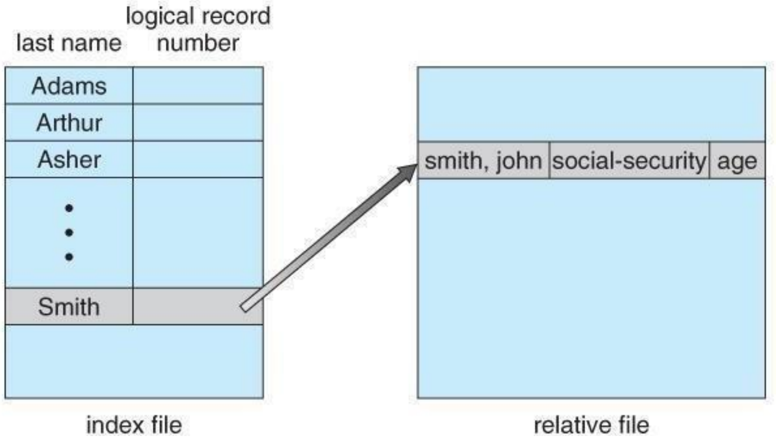
sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;

write_next	write cp;
	cp = cp + 1;

Simulation of sequential access on a direct-access file.

Other Access Methods

- An indexed access scheme can be easily built on top of a direct access system. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes.

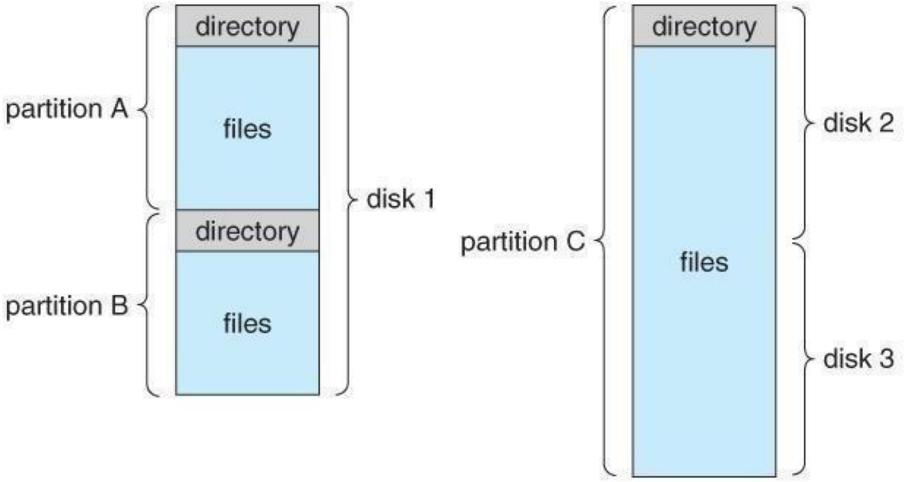


Example of index and relative files.

Directory Structure

Storage Structure

- A disk can be used in its entirety for a file system.
- Alternatively a physical disk can be broken up into multiple ***partitions, slices, or mini-disks***, each of which becomes a virtual disk and can have its own filesystem. (or be used for raw storage, swap space, etc.)
- Or, multiple physical disks can be combined into one ***volume***, i.e. a larger virtual disk, with its own filesystem spanning the physical disks.



A typical file-system organization.

Directory Overview

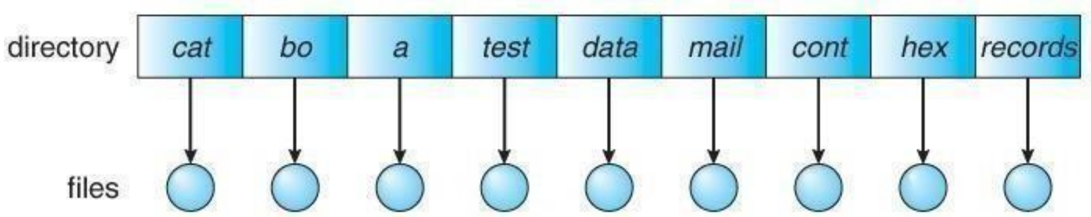
- Directory operations to be supported include:

○ Search for a file
○ Create a file - add to the directory
○ Delete a file - erase from the directory
○ List a directory - possibly ordered in different ways.
○ Rename a file - may change sorting order

○ Traverse the file system.

Single-Level Directory

- Simple to implement, but each file must have a unique name.



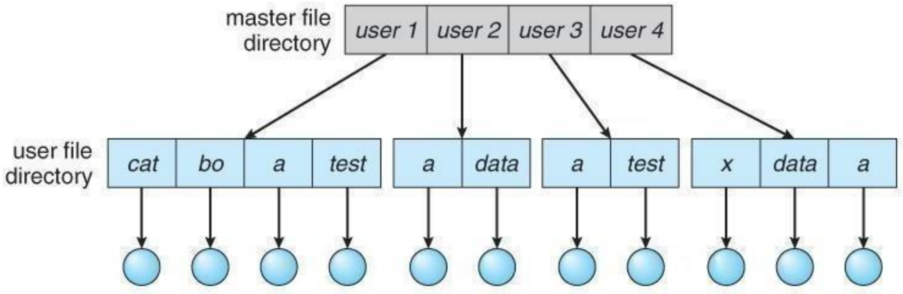
Single-level directory.

Two-Level Directory

- Each user gets their own directory space.
- File names only need to be unique within a given user's directory.
- A master file directory is used to keep track of each users directory, and must be maintained when users are added to or removed from the system.
- A separate directory is generally needed for system (executable) files.
- Systems may or may not allow users to access other directories besides their own

○ If access to other directories is allowed, then provision must be made to specify the directory being accessed.

○ If access is denied, then special consideration must be made for users to run programs located in system directories. A ***search path*** is the list of directories in which to search for executable programs, and can be set uniquely for each user.

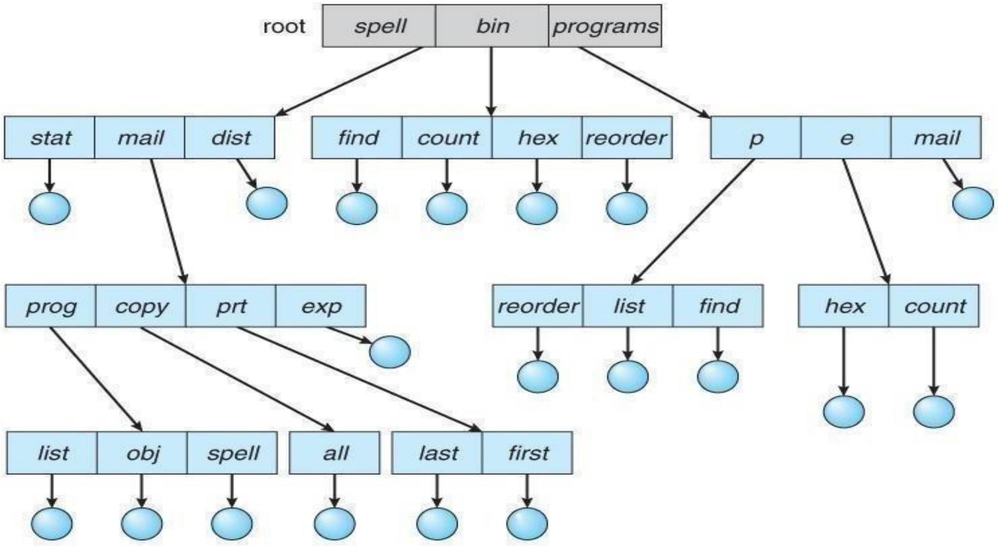


Two-level directory structure.

Tree-Structured Directories

- An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar.
- Each user / process has the concept of a ***current directory*** from which all (relative) searches take place.
- Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.)
- Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.

- One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub- trees.



Tree-structured directory structure.

Acyclic-Graph Directories

- When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic- graph structure. (Note the ***directed*** arcs from parent to child.)
- UNIX provides two types of ***links*** for implementing the acyclic-graph structure. (See "man ln" for more details.)

o A ***hard link*** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same filesystem.

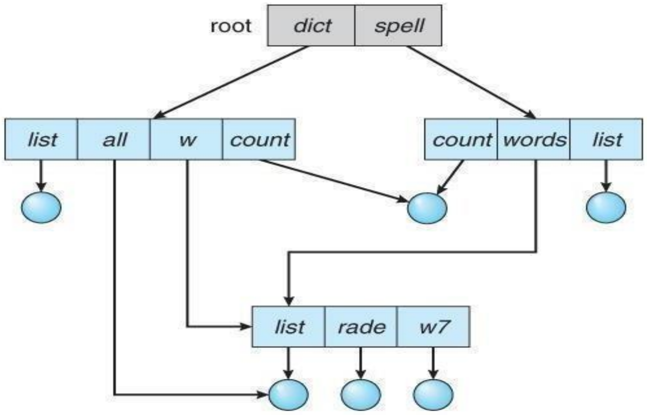
o A ***symbolic link***, that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other filesystems, as well as ordinary files in the current filesystem.

- Windows only supports symbolic links, termed ***shortcuts.***
- Hard links require a ***reference count***, or ***link count*** for each file, keeping track of how many directory entries are currently referring to this file. Whenever one ofthe references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.
- For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:

o One option is to find all the symbolic links and adjust them also.

o Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.

o What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?



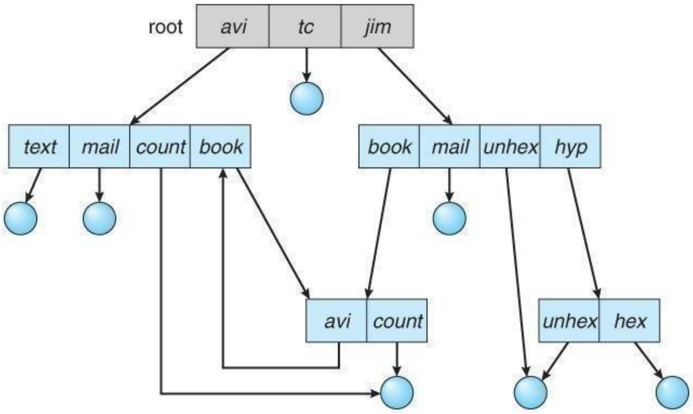
Acyclic-graph directory structure.

General Graph Directory

- If cycles are allowed in the graphs, then several problems can arise:

○ Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories.)

○ Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem. (chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either system. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely deleted.)

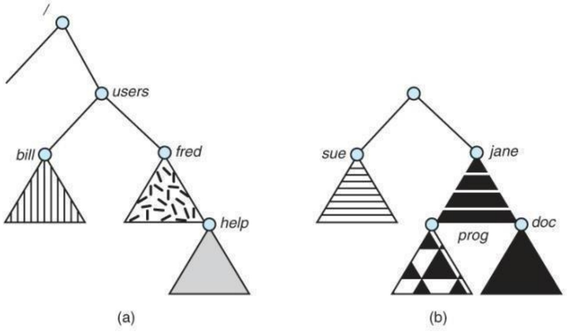


General graph directory.

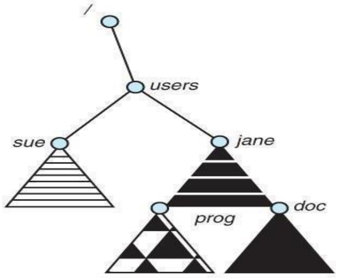
File-System Mounting

- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a filesystem to mount and a ***mount point*** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.
- Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new filesystem are now hidden by the mounted filesystem, and are no longer available. For this reason some systems only allow mounting onto empty directories.

- Filesystems can only be mounted by root, unless root has previously configured certain filesystems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy filesystems to /mnt or something like it.) Anyone can run the mount command to see what filesystems are currently mounted.
- Filesystems may be mounted read-only, or have other restrictions imposed.



File system. (a) Existing system. (b) Unmounted volume.



Mount point.

- The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.
- Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.
- More recent Windows systems allow filesystems to be mounted to any directory in the filesystem, much like UNIX.

File Sharing

Multiple Users

- On a multi-user system, more information needs to be stored for each file:
  - The owner (user) who owns the file, and who can control its access.
  - The group of other user IDs that may have some special access to the file.
  - What access rights are afforded to the owner (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)
  - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

Remote File Systems

- The advent ofthe Internet introduces issues for accessing files stored on remote computers
  - The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account and password controlled, or ***anonymous***, not requiring any user name or password.
  - Various forms of ***distributed file systems*** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
  - The WWW has made it easyonce again to access files on remote systems without mounting their filesystems, generally using (anonymous) ftp as the underlying file transport mechanism.

The Client-Server Model

- When one computer system remotely mounts a filesystem that is physically located on another system, the system which physically owns the files acts as a ***server***, and the system which mounts them is the ***client.***
- User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)
- The same computer can be both a client and a server. (E.g. cross-linked file systems.)
- There are a number of security concerns involved in this model:
  - Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
  - Servers may restrict remote access to read-only.
  - Servers restrict which filesystems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.
- The NFS (Network File System) is a classic example of such a system.

Distributed Information Systems

- The ***Domain Name System, DNS,*** provides for a unique naming system across all of the Internet.
- Domain names are maintained by the ***Network Information System, NIS***, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.
- Microsoft's ***Common Internet File System, CIFS***, establishes a ***network login*** for each user on a networked system with shared file access. Older Windows systems used ***domains***, and newer systems (XP, 2000), use ***active directories.*** User names must match across the network for this system to be valid.
- A newer approach is the ***Lightweight Directory-Access Protocol, LDAP,*** which provides a ***secure single sign-on*** for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

Failure Modes

- When a local disk file is unavailable, the result is generally known immediately, and is generally non- recoverable. The only reasonable response is for the response to fail.
- However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.

Consistency Semantics

- ***Consistency Semantics*** deals with the consistency between the views of shared files on a networked system. When one user changes the file, when do other users see the changes?

UNIX Semantics

- The UNIX file system uses the following semantics:
  - Writes to an open file are immediately visible to any other user who has the file open.
  - One implementation uses a shared location pointer, which is adjusted for all sharing users.
- The file is associated with a single exclusive physical resource, which may delay some accesses.

Session Semantics

- The Andrew File System, AFS uses the following semantics:
  - Writes to an open file are not immediately visible to other users.
  - When a file is closed, any changes made become available only to users who open the file at a later time.
- According to these semantics, a file can be associated with multiple (possibly different) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.
- AFS file systems may be accessible by systems around the world. Access control is maintained through (somewhat) complicated access control lists, which may grant access to the entire world (literally) or to specifically named users accessing the files from specifically named remote environments.

Immutable-Shared-Files Semantics

- Under this system, when a file is declared as ***shared*** by its creator, it becomes immutable and the name cannot be re-used for any other resource. Hence it becomes read-only, and shared access is simple.

Protection

- Files must be kept safe for reliability (against accidental damage), and protection (against deliberate malicious access.) The former is usually managed with backup copies. This section discusses the latter.
- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

Types of Access

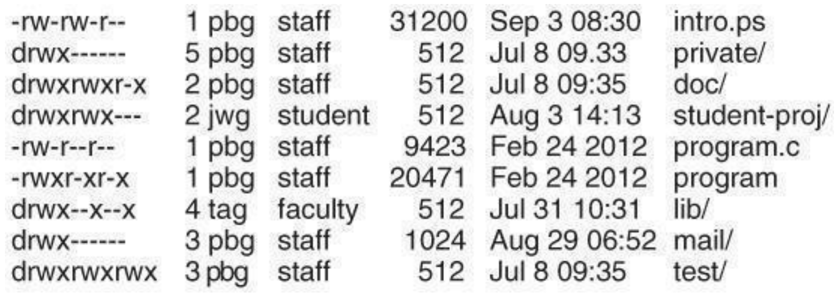
- The following low-level operations are often controlled:
  - Read - View the contents of the file
  - Write - Change the contents ofthe file.
  - Execute - Load the file onto the CPU and follow the instructions contained therein.
    - Append - Add to the end ofan existing file.
    - Delete - Remove a file from the system.
    - List -View the name and other attributes of files on the system.
  - Higher-level operations, such as copy, can generally be performed through combinations of the above.

Access Control

- One approach is to have complicated ***Access Control Lists, ACL,*** which specify exactly what access is allowed or denied for specific users or groups.
  - The AFS uses this system for distributed access.
  - Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)
- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. (See "man chmod" for full details.) The RWX bits control the following privileges for ordinary files and directories:

bit	**Files**	**Directories**
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
	Write (change) file contents.	Change directory contents. Required to create or delete files.
X	Execute file contents as a program.	Access detailed directory information. Required to get a long listing, or to access any specific file in the directory. Note that if a user has X but not R permissions on a directory, they can still access specific files, but only if they already know the name of the file they are trying to access.

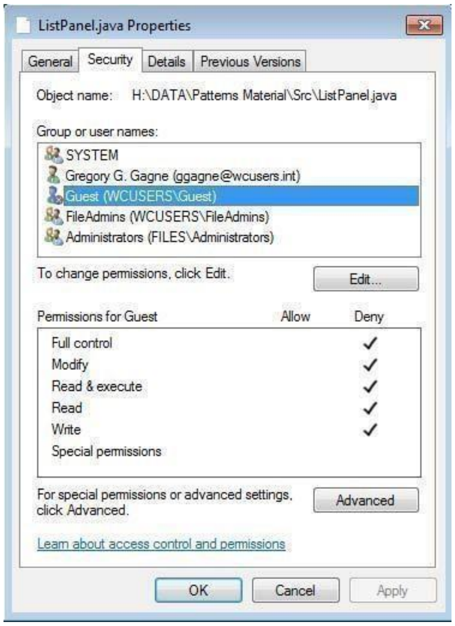
- In addition there are some special bits that can also be applied:
  - The set user ID (SUID) bit and/or the set group ID (SGID) bits applied to executable files temporarily change the identity of whoever runs the program to match that of the owner / group of the executable program. This allows users running specific programs to have access to files (***while running that program***) to which they would normally be unable to access. Setting of these two bits is usually restricted to root, and must be done with caution, as it introduces a potential security leak.
  - The sticky bit on a directory modifies write permission, allowing users to only delete files for which they are the owner. This allows everyone to create files in /tmp, for example, but to only delete files which they have created, and not anyone else's.
  - The SUID, SGID, and sticky bits are indicated with an S, S, and T in the positions for execute permission for the user, group, and others, respectively. If the letter is lower case, (s, s, t), then the corresponding execute permission is not also given. If it is upper case, (S, S, T), then the corresponding execute permission IS given.
  - The numeric form of chmod is needed to set these advanced bits.



```
-rw-rw-r--   1 pbg  staff    31200  Sep 3 08:30   intro.ps
drwx------   5 pbg  staff      512  Jul 8 09.33   private/
drwxrwxr-x   2 pbg  staff      512  Jul 8 09:35   doc/
drwxrwx---   2 jwg  student    512  Aug 3 14:13   student-proj/
-rw-r--r--   1 pbg  staff     9423  Feb 24 2012   program.c
-rwxr-xr-x   1 pbg  staff    20471  Feb 24 2012   program
drwx--x--x   4 tag  faculty    512  Jul 31 10:31  lib/
drwx------   3 pbg  staff     1024  Aug 29 06:52  mail/
drwxrwxrwx   3 pbg  staff      512  Jul 8 09:35   test/
```

Sample permissions in a UNIX system.

- Windows adjusts files access through a simple GUI:


ListPanel.java Properties
General
Security
Details
Previous Versions
To change permissions, click Edit.
Edit...
For special permissions or advanced settings, click Advanced.
Advanced
Learn about access control and permissions
OK
Cancel
Apply

Object name: H:\DATA\Patterns Material\Src\ListPanel.java

Group or user names:

SYSTEM
Gregory G. Gagne (ggagne@wcusers.int)
Guest (WCUSERS\Guest)
FileAdmins (WCUSERS\FileAdmins)
Administrators (FILES\Administrators)

Permissions for Guest	Allow	Deny
Full control		✓
Modify		✓
Read & execute		✓
Read		✓
Write		✓
Special permissions		

Windows 7 access-control list management.

Other Protection Approaches and Issues

- Some systems can apply passwords, either to individual files, or to specific sub-directories, or to the entire system. There is a trade-off between the number of passwords that must be maintained (and remembered by the users) and the amount of information that is vulnerable to a lost or forgotten password.
- Older systems which did not originally have multi-user file access permissions (DOS and older versions of Mac) must now be ***retrofitted*** if they are to share files on a network.
- Access to a file requires access to all the files along its path as well. In a cyclic directory structure, users may have different access to the same file accessed through different paths.
- Sometimes just the knowledge of the existence of a file of a certain name is a security (or privacy) concern. Hence the distinction between the R and X bits on UNIX directories.

FILE-SYSTEM STRUCTURE

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency.
- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:
  - At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more ofthe electronic

controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.

- ***I/O Control*** consists of ***device drivers***, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ***ports***) that it listens to, and a unique set of command codes and results codes that it understands.
- The ***basic file system*** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.
- The ***file organization module*** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
- The ***logical file system*** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the meta data as well as block number information for finding the data on the disk.

- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be filesystem specific. Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 (among 40 others supported.)

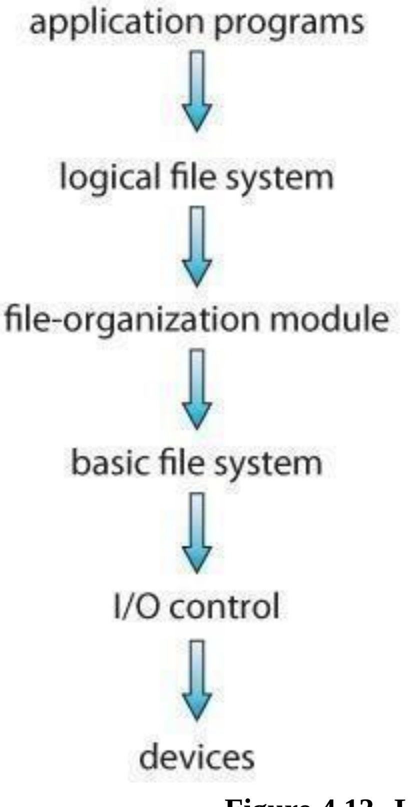
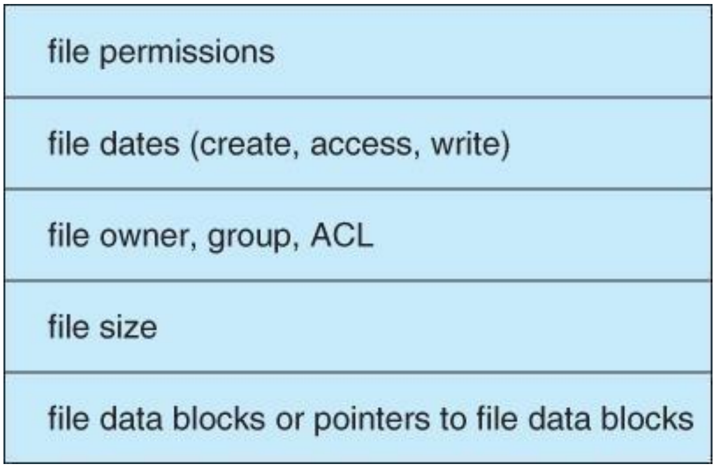


Figure 4.13- Layered file system.

FILE-SYSTEM IMPLEMENTATION

Overview

- File systems store several important data structures on the disk:
  - A ***boot-control block***, (per volume) a.k.a. the ***boot block*** in UNIX or the ***partition boot sector*** in Windows contains information about how to boot the system off of this disk. This will generally be the first sector ofthe volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
  - A ***volume control block,*** (per volume) a.k.a. the ***master file table*** in UNIX or the ***superblock*** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.
  - A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a ***master file table.***
  - The ***File Control Block, FCB,*** (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.



file permissions

file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

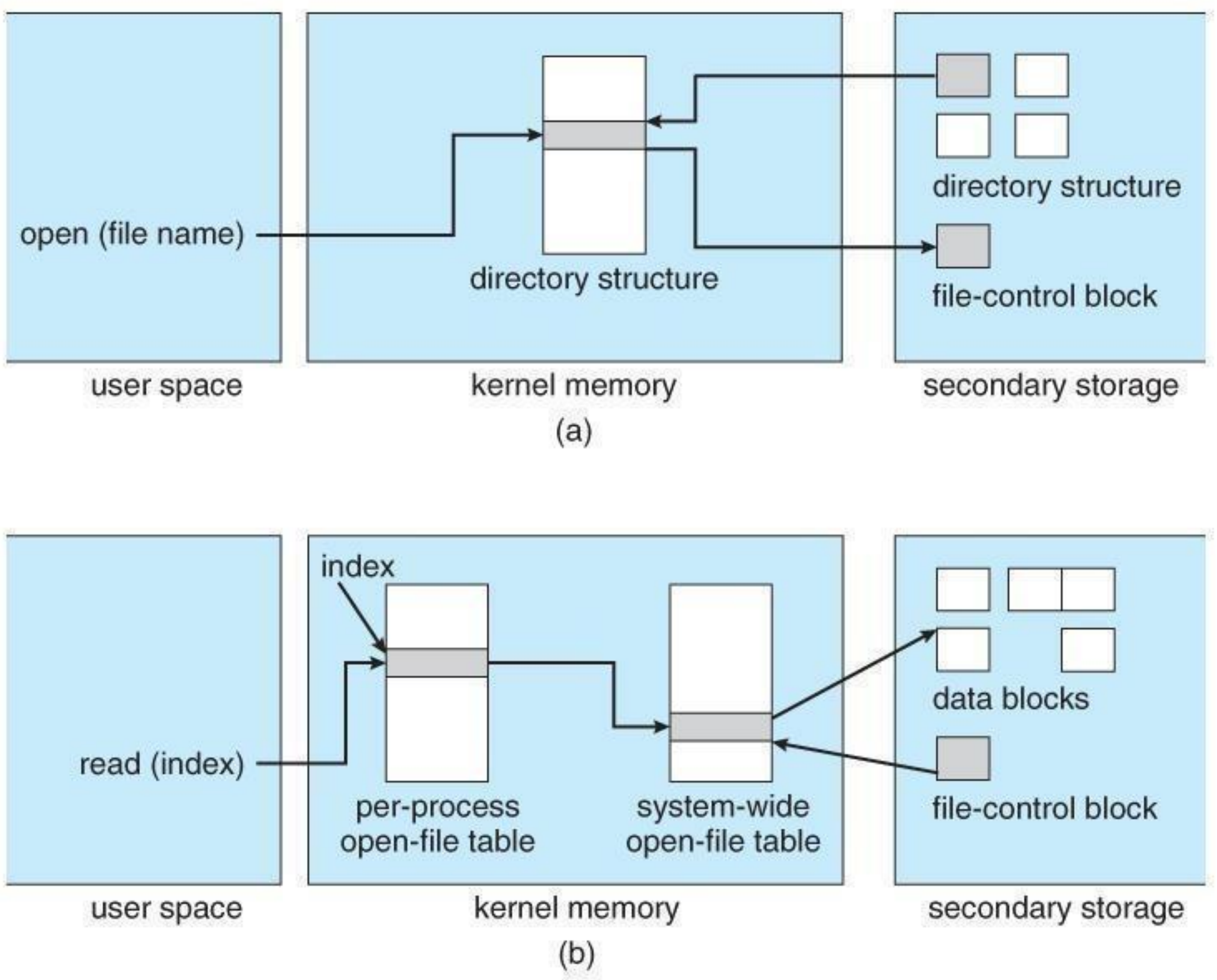
A typical file-control block.

- There are also several key data structures stored in memory:
  - An in-memory mount table.
  - An in-memory directory cache of recently accessed directory information.
  - ***A system-wide open file table***, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
  - ***A per-process open file table,*** containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)
- Figure below illustrates some of the interactions of file system components when files are created and/or used:
  - When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
  - When a file is accessed during a program, the open( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table

referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a ***file descriptor***, and Windows refers to it as a ***file handle***.

o If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.

o When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.



- In-memory file-system structures. (a) File open. (b) File read.

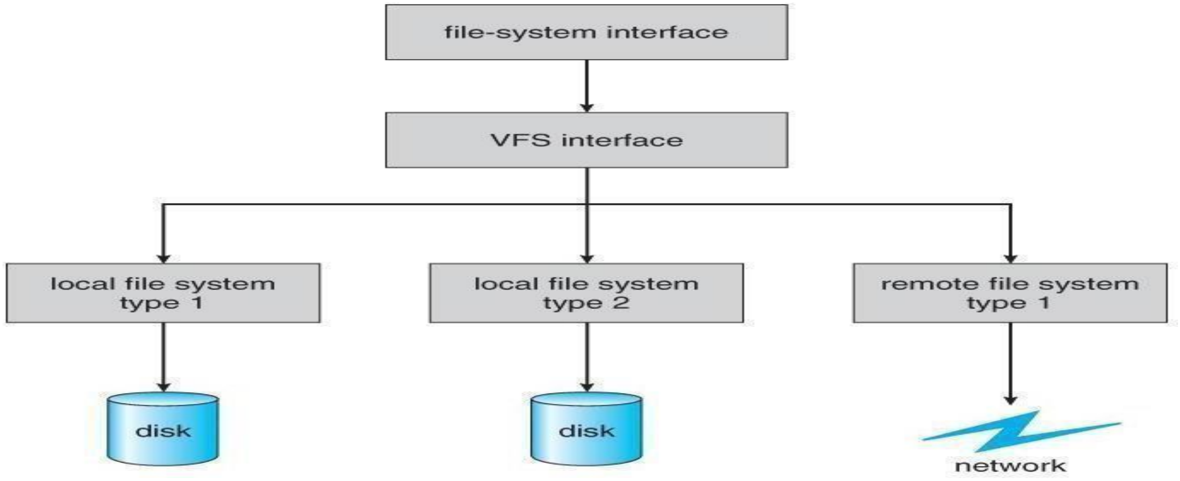
Partitions and Mounting

- Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.
- Partitions can either be used as raw devices (with no structure imposed upon them), or they can be formatted to hold a filesystem (i.e. populated with FCBs and initial directory structures as appropriate.) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.

- The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. Modern boot programs understand multiple OSes and filesystem formats, and can give the user a choice of which of several available systems to boot.
- The ***root partition*** contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. (Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.)
- Continuing with the boot process, additional filesystems get mounted, adding their information into the appropriate mount table structure. As a part of the mounting process the file systems may be checked for errors or inconsistencies, either because they are flagged as not having been closed properly the last time they were used, or just for general principals. Filesystems may be mounted either automatically or manually. In UNIX a mount point is indicated by setting a flag in the in-memory copy of the inode, so all future references to that inode get re-directed to the root directory of the mounted filesystem.

Virtual File Systems

- ***Virtual File Systems, VFS***, provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier (vnode) for files across the entire space, including across all filesystems of different types. (UNIX inodes are unique only across a single filesystem, and certainly do not carry across networked file systems.)
- The VFS in Linux is based upon four key object types:
  - The ***inode*** object, representing an individual file
  - The ***file*** object, representing an open file.
  - The ***superblock*** object, representing a filesystem.
  - The ***dentry*** object, representing a directory entry.
- Linux VFS provides a set of common functionalities for each filesystem, using function pointers accessed through a table. The same functionality is accessed through the same table position for all filesystem types, though the actual functions pointed to by the pointers may be filesystem-specific. See /usr/include/linux/fs.h for full details. Common operations provided include open( ), read( ), write( ), and mmap( ).



Schematic view of a virtual file system.

DIRECTORY IMPLEMENTATION

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- Finding a file (or verifying one does not already exist upon creation) requires a linear search.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- Sorting the list makes searches faster, at the expense of more complex insertions and deletions.
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- More complex data structures, such as B-trees, could also be considered.

Hash Table

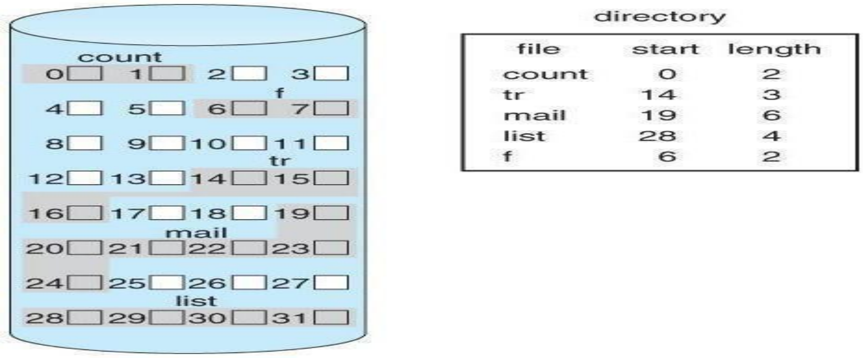
- A hash table can also be used to speed up searches.
- Hash tables are generally implemented ***in addition to*** a linear or other structure

ALLOCATION METHODS

- There are three major methods of storing files on disks: contiguous, linked, and indexed.

Contiguous Allocation

- ***Contiguous Allocation*** requires that all blocks of a file be kept together contiguously.
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.
- Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.
- (Even file systems that do not by default store files contiguously can benefit from certain utilities that compact the disk and make all files contiguous in the process.)
- Problems can arise when files grow, or if the exact size ofa file is unknown at creation time:
  - Over-estimation ofthe file's final size increases external fragmentation and wastes disk space.
  - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
  - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.
- A variation is to allocate file space in large contiguous chunks, called ***extents.*** When a file outgrows its original extent, then an additional one is allocated. (For example an extent may be the size of a complete track or even cylinder, aligned on an appropriate track or cylinder boundary.) The high- performance files system Veritas uses extents to optimize performance.

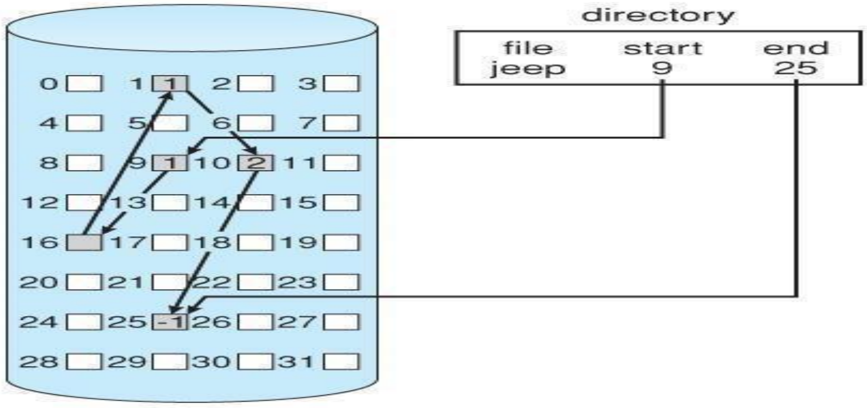


file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation of disk space.

Linked Allocation

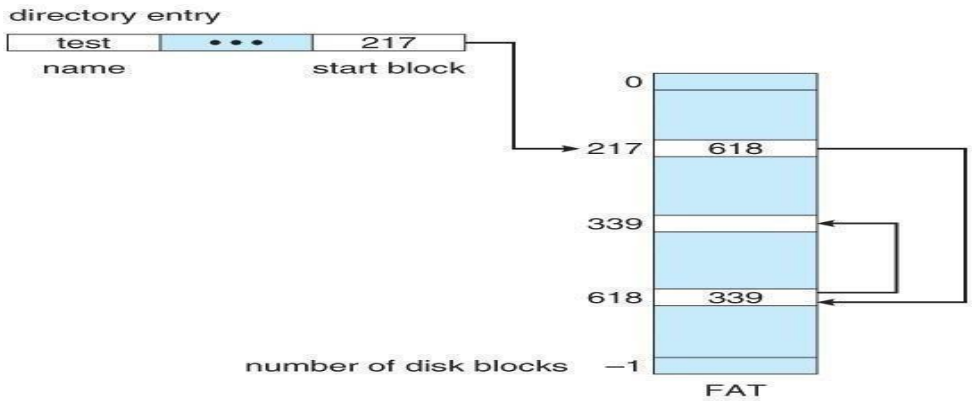
- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.
- Allocating ***clusters*** of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.



file	start	end
jeep	9	25

Linked allocation of disk space.

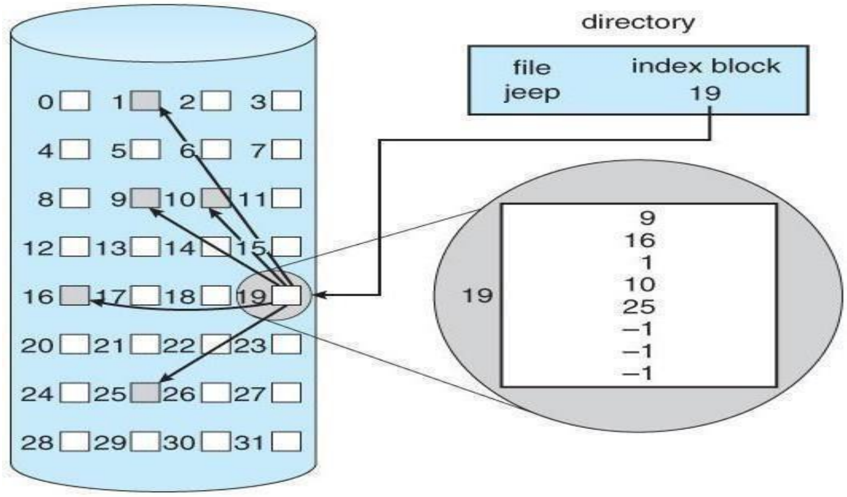
The ***File Allocation Table, FAT,*** used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.



File-allocation table.

Indexed Allocation

- ***Indexed Allocation*** combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.



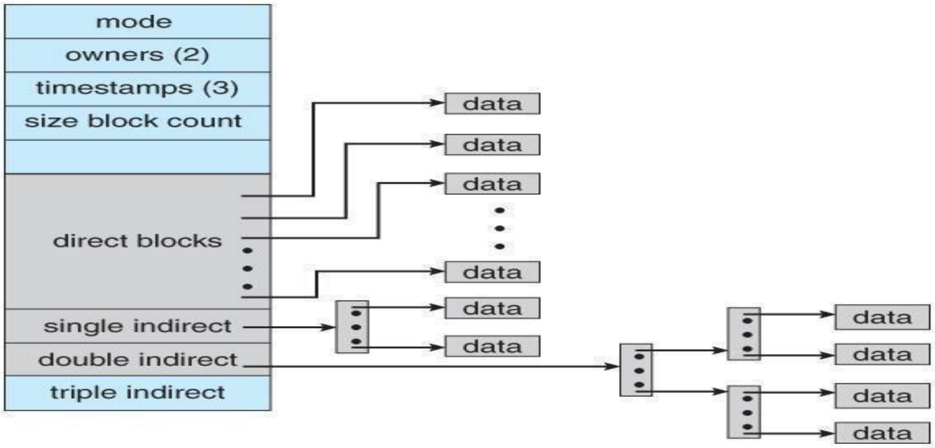
Indexed allocation of disk space.

- Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

○ **Linked Scheme -** An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.

○ **Multi-Level Index -** The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.

○ **Combined Scheme -** This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below.) The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)



The UNIX inode.

Performance

- The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.
- Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities.
- Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.
- And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

FREE-SPACE MANAGEMENT

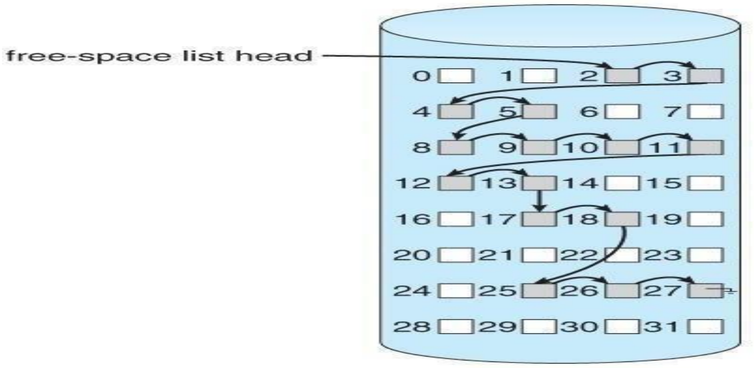
- Another important aspect of disk management is keeping track of and allocating free space.

Bit Vector

- One simple approach is to use a ***bit vector***, in which each bit represents a disk block, set to 1 if free or 0 if allocated.
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- The down side is that a 40GB disk requires over 5MB just to store the bitmap. (For example.)

Linked List

- A linked list can also be used to keep track of all free blocks.
- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
- The FAT table keeps track ofthe free list as just one more linked list on the table.



Linked free-space list on disk.

Grouping

A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

Counting

- When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

Space Maps

- Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) ***metaslabs*** of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

EFFICIENCY AND PERFORMANCE

Efficiency

- UNIX pre-allocates inodes, which occupies space even before any files are created.
- UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.
- Some systems use variable size clusters depending on the file size.
- The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written.
- As technology advances, addressing schemes have had to grow as well.

○ Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^128 bytes with atomic storage would be at least 272 trillion kilograms!)

- Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

Performance

- Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing latency.) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.
- Some OSes cache disk blocks they expect to need again in a ***buffer cache.***
- A ***page cache*** connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.
- Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a ***unified virtual memory.***
- Figures below show the advantages of the ***unified buffer cache*** found in some versions of UNIX and Linux - Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.

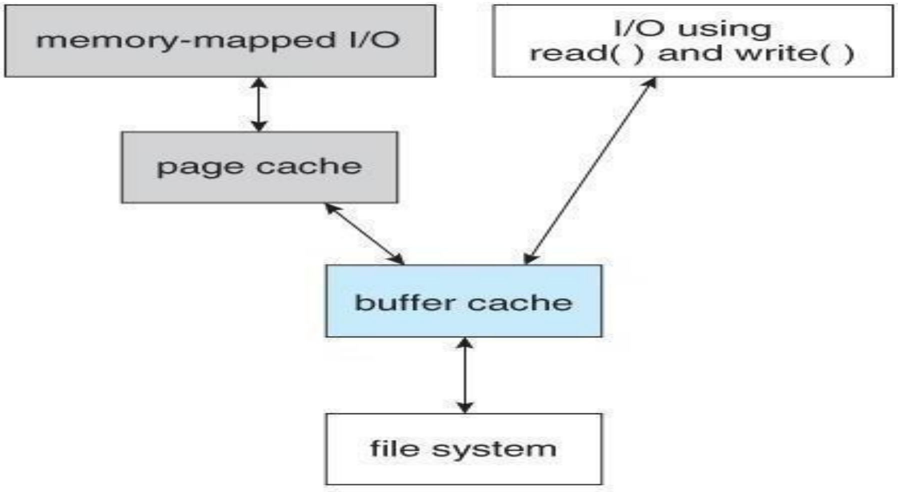


Figure - I/O without a unified buffer cache.

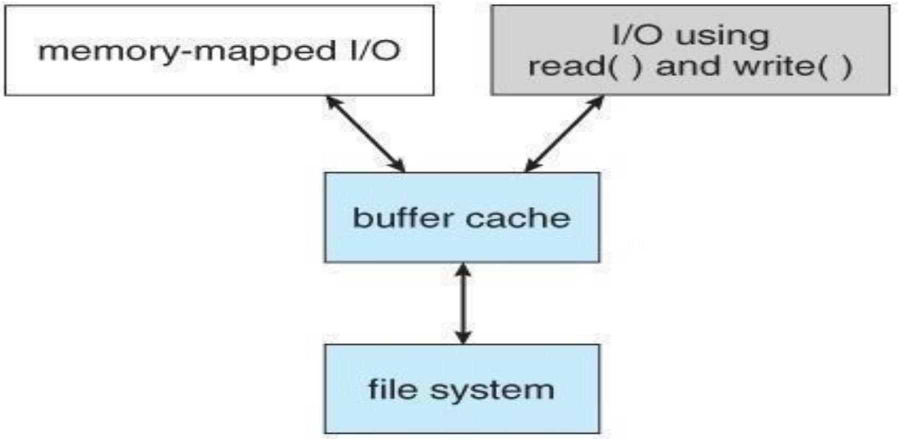


Figure I/O using a unified buffer cache.

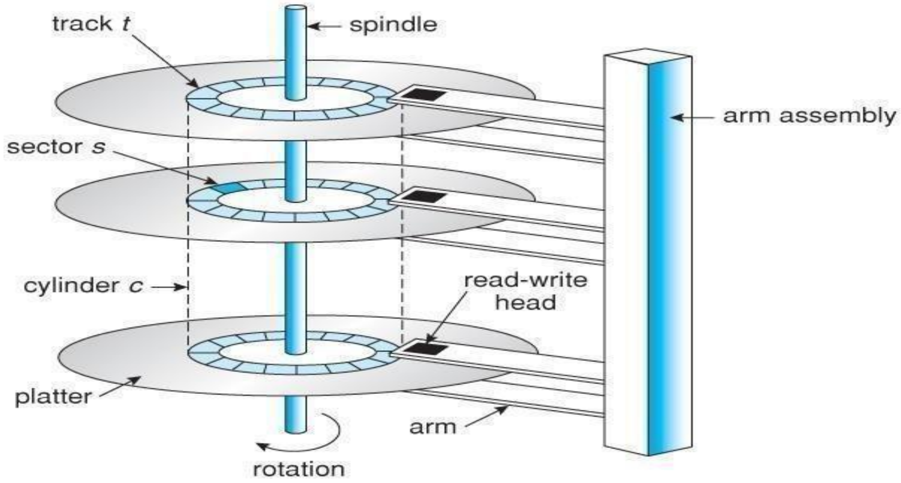
- Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in ***priority paging*** giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.
- Another issue affecting performance is the question of whether to implement ***synchronous writes*** or ***asynchronous writes.*** Synchronous writes occur in the order in which the disk subsystem receives them, without caching; Asynchronous writes are cached, allowing the disk subsystem to schedule writes in a more efficient order (See Chapter 12.) Metadata writes are often done synchronously. Some systems support flags to the open call requiring that writes be synchronous, for example for the benefit of database systems that require their writes be performed in a required order.
- The type of file access can also have an impact on optimal page replacement policies. For example, LRU is not necessarily a good policy for sequential access files. For these types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, (if it is ever needed at all.) On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:
  - ***Free-behind*** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
  - ***Read-ahead*** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of transferring data from the disk controller memory into motherboard main memory.
- The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times. (See Chapter 12.) Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.

MASS-STORAGE STRUCTURE

Overview of Mass-Storage Structure

Magnetic Disks

- Traditional magnetic disks have the following basic structure:
  - One or more ***platters*** in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal, while "***floppy***" disks are made of more flexible plastic.
  - Each platter has two working ***surfaces.*** Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.
  - Each working surface is divided into a number of concentric rings called ***tracks.*** The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a ***cylinder***.
  - Each track is further divided into ***sectors,*** traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)
  - The data on a hard drive is read by read-write ***heads.*** The standard configuration (shown below) uses one head per surface, each on a separate ***arm***, and controlled by a common ***arm assembly*** which moves all heads simultaneously from one cylinder to another. (Other configurations, including independent read- write heads, may speed up disk access, but involve serious technical difficulties.)
  - The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head -sector- cylinder number at which it is located.



-Moving-head disk mechanism.

- In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:
  - The ***positioning time***, a.k.a. the ***seek time*** or ***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.
  - The ***rotational latency*** is the amount of time required for the desired sector to rotate around and come under the read-write head.This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time. (For a disk rotating at 7200 rpm, the average rotational latency would be 1/2 revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.

- The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer. (Some authors may also use the term transfer rate to refer to the overall transfer rate, inclu ding seek time and rotational latency as well as the electronic data transfer rate.)

- Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a ***head crash*** occurs, which may or may not permanently damage the disk or even destroy it completely. For this reason it is normal to ***park*** the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.
- Floppy disks are normally ***removable***. Hard drives can also be removable, and some are even ***hot-swappable***, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.
- Disk drives are connected to the computer via a cable known as the ***I/O Bus.*** Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.
- The ***host controller*** is at the computer end of the I/O bus, and the ***disk controller*** is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred betweeen the magnetic surface and onboard ***cache*** by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

Solid-State Disks - New

- As technologies improve and economics change, old technologies are often used in different ways. One example of this is the increasing used of ***solid state disks, or SSDs.***
- SSDs use memory technology as a small fast hard disk. Specific implementations may use either flash memory or DRAM chips protected by a battery to sustain the information through power cycles.
- Because SSDs have no moving parts they are much faster than traditional hard drives, and certain problems such as the scheduling of disk accesses simply do not apply.
- However SSDs also have their weaknesses: They are more expensive than hard drives, generally not as large, and may have shorter life spans.
- SSDs are especially useful as a high-speed cache of hard-disk information that must be accessed quickly. One example is to store filesystem meta-data, e.g. directory and inode information, that must be accessed quickly and often. Another variation is a boot disk containing the OS and some application executables, but no vital user data. SSDs are also used in laptops to make them smaller, faster, and lighter.
- Because SSDs are so much faster than traditional hard disks, the throughput of the bus can become a limiting factor, causing some SSDs to be connected directly to the system PCI bus for example.

Magnetic Tapes - Magnetic tapes were once used for common secondary storage before the days of

hard disk drives, but today are used primarily for backups.

- Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives.
- Capacities oftape drives can range from 20 to 200 GB, and compression can double that capacity.

Disk Structure

- The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:

1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.
3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.

- There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.
- Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:
  - With ***Constant Linear Velocity, CLV,*** the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
  - With ***Constant Angular Velocity, CAV,*** the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

DISK ATTACHMENT

Disk drives can be attached either directly to a particular host (a local disk) or to a network.

Host-Attached Storage

- Local disks are accessed through I/O Ports as described earlier.
- The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller.
- SATA is similar with simpler cabling.
- High end workstations or other systems in need of larger number of disks typically use SCSI disks:
  - The SCSI standard supports up to 16 ***targets*** on each SCSI bus, one of which is generally the host adapter and the other 15 of which can be disk or tape drives.
  - A SCSI target is usually a single drive, but the standard also supports up to 8 ***units*** within each target. These would generally be used for accessing individual disks within a RAID array. (See below.)
  - The SCSI standard also supports multiple host adapters in a single computer, i.e. multiple SCSI busses.
  - Modern advancements in SCSI include "fast" and "wide" versions, as well as SCSI-2.
  - SCSI cables may be either 50 or 68 conductors. SCSI devices may be external as well as internal.
  - See wikipedia for more information on the SCSI interface.
- FC is a high-speed serial architecture that can operate over optical fiber or four-conductor copper wires, and has two variants:
  - A large switched fabric having a 24-bit address space. This variant allows for multiple devices and multiple hosts to interconnect, forming the basis for the ***storage-area networks, SANs,*** to be discussed in a future section.

- The ***arbitrated loop, FC-AL,*** that can address up to 126 devices (drives and controllers.)

Network-Attached Storage

- Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS filesystem mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage.
- NAS can be implemented using SCSI cabling, or ***ISCSI*** uses Internet protocols and standard network connections, allowing long-distance remote access to shared files.
- NAS allows computers to easily share data storage, but tends to be less efficient than standard host- attached storage.

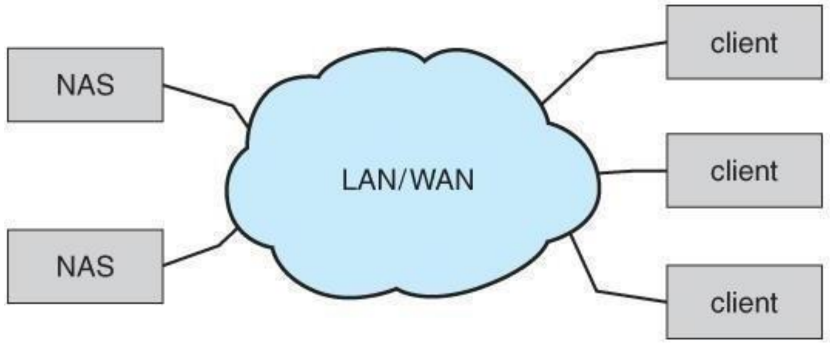


Figure 4.26- Network-attached storage.

Storage-Area Network

- A ***Storage-Area Network, SAN,*** connects computers and storage devices in a network, using storage protocols instead of network protocols.
- One advantage ofthis is that storage access does not tie up regular networking bandwidth.
- SAN is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly.
- SAN is also controllable, allowing restricted access to certain hosts and devices.

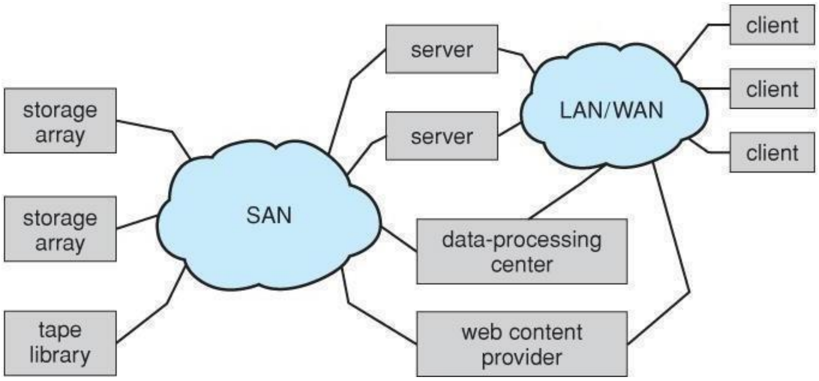


Figure - Storage-area network.

DISK SCHEDULING

- As mentioned earlier, disk transfer speeds are limited primarily by ***seek times*** and ***rotational latency.*** When multiple requests are to be processed there is also some inherent delay in waiting for other requests to be processed.
- ***Bandwidth*** is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed, (for a series of disk requests.)
- Both bandwidth and access time can be improved by processing requests in a good order.
- Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing.

FCFS Scheduling

- ***First-Come First-Serve*** is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:

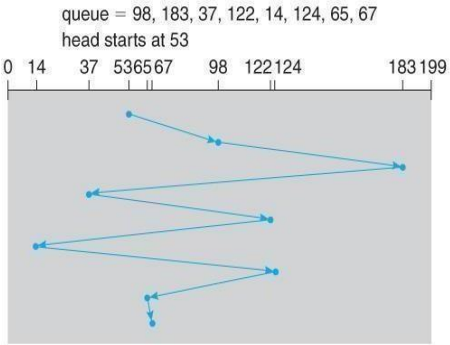


Figure -FCFS disk scheduling.

SSTF Scheduling

- ***Shortest Seek Time First*** scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk.
- SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.

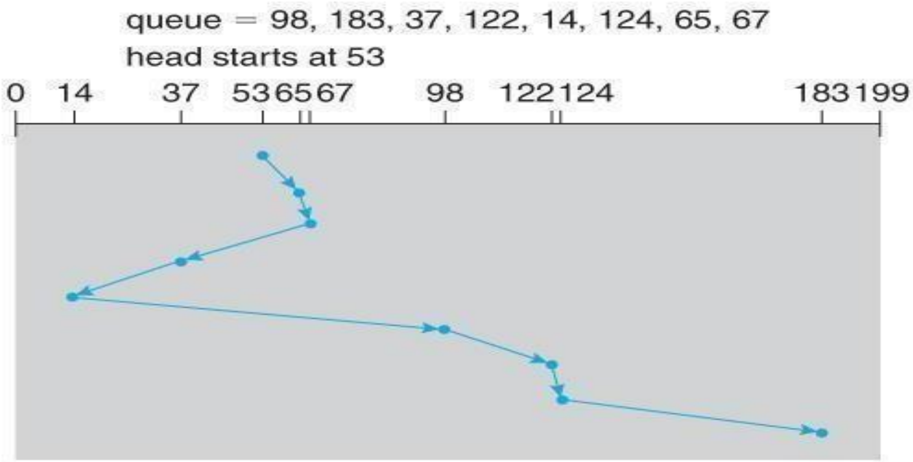


Figure - SSTF disk scheduling.

SCAN Scheduling

- The ***SCAN*** algorithm, a.k.a. the ***elevator*** algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.

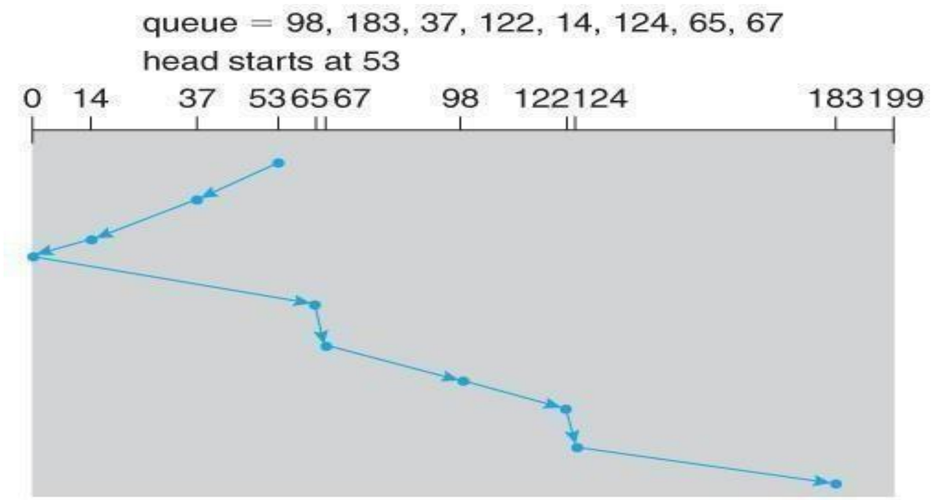


Figure - SCAN disk scheduling.

- Under the SCAN algorithm, If a request arrives just ahead of the moving head then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a fairly wide variation in access times which can be improved upon.
- Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.

C-SCAN Scheduling

- The ***Circular-SCAN*** algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk:

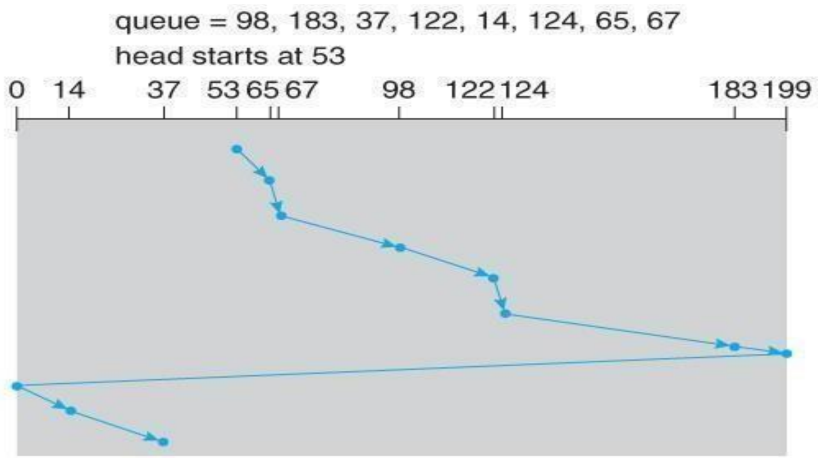


Figure - C-SCAN disk scheduling.

LOOK Scheduling

- ***LOOK*** scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:

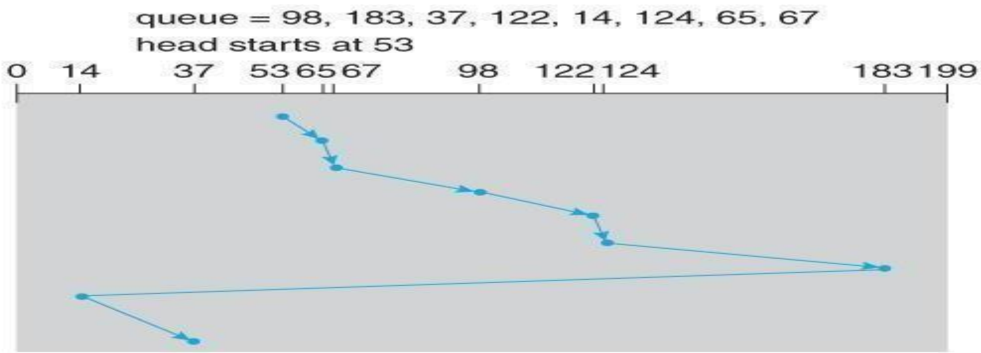


Figure 4.32- C-LOOK disk scheduling.

Selection of a Disk-Scheduling Algorithm

- With very low loads all algorithms are equal, since there will normally only be one request to process at a time.
- For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough.
- For busier systems, SCAN and LOOK algorithms eliminate starvation problems.
- The actual optimal algorithm may be something even more complex than those discussed here, but the incremental improvements are generally not worth the additional overhead.
- Some improvement to overall filesystem access times can be made by intelligent placement of directory and/or inode information. If those structures are placed in the middle of the disk instead of at the beginning of the disk, then the maximum distance from those structures to data blocks is reduced to only one-half of the disk size. If those structures can be further distributed and furthermore have their data blocks stored as close as possible to the corresponding directory structures, then that reduces still further the overall time to find the disk block numbers and then access the corresponding data blocks.

- On modern disks the rotational latency can be almost as significant as the seek time, however it is not within the OSes control to account for that, because modern disks do not reveal their internal sector mapping schemes, (particularly when bad blocks have been remapped to spare sectors.)
    - Some disk manufacturers provide for disk scheduling algorithms directly on their disk controllers, (which do know the actual geometry of the disk as well as any remapping), so that if a series of requests are sent from the computer to the controller then those requests can be processed in an optimal order.
    - Unfortunately there are some considerations that the OS must take into account that are beyond the abilities of the on-board disk-scheduling algorithms, such as priorities of some requests over others, or the need to process certain requests in a particular order. For this reason OSes may elect to spoon-feed requests to the disk controller one at a time in certain situations.