KarpagamAcademy of Higher Education
(Established Under Section 3 of UGC Act 1956)
Eachanari Post, Coimbatore – 641 021. INDIA
Phone : 0422-2611146, 2611082 Fax No : 0422 -2611043

16CAU601B UNIX/LINUX PROGRAMMING

LECTURE PLAN

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT I			
1	1	What is Linux/Unix Operating Systems	R1:12-15,W1
2	1	Difference between linux/unix and other operating systems	R1:16-20
3	1	Features and Architecture	R2:15-22
4	1	Various Distributions available in the market	R2:23-27
		Installation, Booting and shutdown process	R1:25-30
5	1	Recapitulation and discussion of Important Questions	
Total No. of Hours planned for Unit-I : 5 Hours			

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT II			
1	1	System processes (an overview)	R3:53-58
2	1	External and internal commands	R3:65-69
3	1	Creation of partitions in OS	R3:70-77,W2
4	1	Processes and its creation phases, Fork, Exec, wait	R3:78-86,W3
5	1	Recapitulation and discussion of Important Questions	
Total No. of Hours planned for Unit II : 5 Hours			

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT III			
1	1	Types of Users, Creating users	R4:63-71
2	1	Granting rights User management commands	R4:72-77
3	1	File quota and various file systems available	R4:78-87
4	1	File System Management and Layout, File permissions	R4:88-96,J1
5	1	Login process, Managing Disk Quotas, Links (hard links, symbolic links)	R4:103-112
6	1	Recapitulation and discussion of Important Questions	
Total No. of Hours planned for Unit III : 6 Hours			

LECTURE PLAN

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT IV			
1	1	**Shell introduction and Shell Scripting**	R4:129-138
2	1	What is shell and various type of shell	R4:139-143, W4
3	1	Various editors present in Linux Different modes of operation in vi editor	R4:143-148
4	1	What is shell script, Writing and executing the shell script	R4:149-157
5	1	Shell variable (user defined and system variables)	R4:158-166
6	1	Recapitulation and Discussion of important Questions	
Total No. of Hours planned for Unit IV : 6 Hours			

LECTURE PLAN

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT V			
1	1	**System calls, Using system calls**	R4:167-173
2	1	Pipes and Filters, Decision making in Shell Scripts (If else, switch),	W5,R4:174-177
3	1	Loops in shell, Functions	R4:178-186
4	1	Utility programs (cut, paste, join, tr, uniq utilities), Pattern matching utility (grep)	R4:187-195
5	1	Recapitulation and Discussion of important Questions	
6	1	Discussion of Previous ESE papers	
7	1	Discussion of Previous ESE papers	
8	1	Discussion of Previous ESE papers	
		Total No. of Hours planned for Unit V : 8 Hours	
	TOTAL PLANNED HOURS : 30		

REFERENCE BOOKS:

R1: Michael Jang, (2011). *RHCSA/ RHCE Red Hat Linux Certification: Exams (Ex200 & Ex300)* ,Certification Press.

R2:Nemeth Synder & Hein,(2010). *Linux Administration Handbook*, (2nd ed.) Pearson Education.

R3: Sumitabha, Das, (2006). *Unix Concepts And Applications*, Tata McGraw-Hill Education.

R4: Richard Stevens,W., Bill Fenner, Andrew M. Rudoff, (2014). *Unix Network Programming, The sockets Networking,* Vol. 1, 3rd ed. API.

WEBSITES

W1:http://en.wikipedia.org/wiki/Linux

W2:http://en.wikibooks.org/wiki/partitions_unix

W3:www.tutorialspoint.com/fork

W4: www.guru99.com/introduction_to_shell_scripting.html

W5: www.tutorialspoint.com/unix/unix_decision_making.htm

JOURNALS:

J1: ”Role of File System in Operating System”, Int. Journal of Computer Science and innovation, Vol 3, 2016.

Enable | Enlighten | Enrich
KARPAGAM
ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)

KarpagamAcademy of Higher Education
(Established Under Section 3 of UGC Act 1956)
Eachanari Post, Coimbatore – 641 021. INDIA
Phone : 0422-2611146, 2611082 Fax No : 0422 -2611043

16CAU601B UNIX/LINUX PROGRAMMING

LECTURE PLAN

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT I			
1	1	What is Linux/Unix Operating Systems	R1:12-15,W1
2	1	Difference between linux/unix and other operating systems	R1:16-20
3	1	Features and Architecture	R2:15-22
4	1	Various Distributions available in the market	R2:23-27
		Installation, Booting and shutdown process	R1:25-30
5	1	Recapitulation and discussion of Important Questions	
Total No. of Hours planned for Unit-I : 5 Hours			

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT II			
1	1	System processes (an overview)	R3:53-58
2	1	External and internal commands	R3:65-69
3	1	Creation of partitions in OS	R3:70-77,W2
4	1	Processes and its creation phases, Fork, Exec, wait	R3:78-86,W3
5	1	Recapitulation and discussion of Important Questions	
Total No. of Hours planned for Unit II : 5 Hours			

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT III			
1	1	Types of Users, Creating users	R4:63-71
2	1	Granting rights User management commands	R4:72-77
3	1	File quota and various file systems available	R4:78-87
4	1	File System Management and Layout, File permissions	R4:88-96,J1
5	1	Login process, Managing Disk Quotas, Links (hard links, symbolic links)	R4:103-112
6	1	Recapitulation and discussion of Important Questions	
Total No. of Hours planned for Unit III : 6 Hours			

LECTURE PLAN

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
UNIT IV			
1	1	**Shell introduction and Shell Scripting**	R4:129-138
2	1	What is shell and various type of shell	R4:139-143, W4
3	1	Various editors present in Linux Different modes of operation in vi editor	R4:143-148
4	1	What is shell script, Writing and executing the shell script	R4:149-157
5	1	Shell variable (user defined and system variables)	R4:158-166
6	1	Recapitulation and Discussion of important Questions	
Total No. of Hours planned for Unit IV : 6 Hours			

LECTURE PLAN

S.No	Lecturer Duration(Hrs)	Topics to be Covered	Support Materials
		UNIT V	
1	1	**System calls, Using system calls**	R4:167-173
2	1	Pipes and Filters, Decision making in Shell Scripts (If else, switch),	W5,R4:174-177
3	1	Loops in shell, Functions	R4:178-186
4	1	Utility programs (cut, paste, join, tr, uniq utilities), Pattern matching utility (grep)	R4:187-195
5	1	Recapitulation and Discussion of important Questions	
6	1	Discussion of Previous ESE papers	
7	1	Discussion of Previous ESE papers	
8	1	Discussion of Previous ESE papers	
		Total No. of Hours planned for Unit V : 8 Hours	
	TOTAL PLANNED HOURS : 30		

REFERENCE BOOKS:

R1: Michael Jang, (2011). *RHCSA/ RHCE Red Hat Linux Certification: Exams (Ex200 & Ex300)* ,Certification Press.

R2:Nemeth Synder & Hein,(2010). *Linux Administration Handbook*, (2nd ed.) Pearson Education.

R3: Sumitabha, Das, (2006). *Unix Concepts And Applications*, Tata McGraw-Hill Education.

R4: Richard Stevens,W., Bill Fenner, Andrew M. Rudoff, (2014). *Unix Network Programming, The sockets Networking,* Vol. 1, 3rd ed. API.

WEBSITES

W1:http://en.wikipedia.org/wiki/Linux

W2:http://en.wikibooks.org/wiki/partitions_unix

W3:www.tutorialspoint.com/fork

W4: www.guru99.com/introduction_to_shell_scripting.html

W5: www.tutorialspoint.com/unix/unix_decision_making.htm

JOURNALS:

J1: ”Role of File System in Operating System”, Int. Journal of Computer Science and innovation, Vol 3, 2016.

Unit-I

Introduction What is Linux/Unix Operating systems, Difference between linux/unix and other operating systems , Features and Architecture, Various Distributions available in the market, Installation, Booting and shutdown process

What is Unix Operating systems

UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session.

What is Linux Operationg System?

Linux is a Unix-like, open source and community-developed operating system for computers, servers, mainframes, mobile devices and embedded devices. It is supported on almost every major computer platform including x86, ARM and SPARC, making it one of the most widely supported operating systems.

Difference between Linux/Unix and other operating systems

	Linux	**Unix**
Cost	Linux can be freely distributed, downloaded freely, distributed through magazines, Books etc. There are priced versions for Linux also, but they are normally cheaper than Windows.	Different flavors of Unix have different cost structures according to vendors
Development and	Linux is developed by Open Source development i.e. through	Unix systems are divided into various other flavors, mostly developed by AT&T as well

	Linux	Unix
Distribution	sharing and collaboration of code and features through forums etc and it is distributed by various vendors.	as various commercial vendors and non-profit organizations.
Manufacturer	Linux kernel is developed by the community. Linus Torvalds oversees things.	Three bigest distributions are Solaris (Oracle), AIX (IBM) & HP-UX Hewlett Packard. And Apple Makes OSX, an unix based os..
User	Everyone. From home users to developers and computer enthusiasts alike.	Unix operating systems were developed mainly for mainframes, servers and workstations except OSX, Which is designed for everyone. The Unix environment and the client-server program model were essential elements in the development of the Internet
Usage	Linux can be installed on a wide variety of computer hardware, ranging from mobile phones, tablet computers and video game consoles, to mainframes and supercomputers.	The UNIX operating system is used in internet servers, workstations & PCs. Backbone of the majority of finance infastructure and many 24x365 high availability solutions.
File system support	Ext2, Ext3, Ext4, Jfs, ReiserFS, Xfs, Btrfs, FAT, FAT32, NTFS	jfs, gpfs, hfs, hfs+, ufs, xfs, zfs format
Text mode interface	BASH (Bourne Again SHell) is the Linux default shell. It can support multiple command interpreters.	Originally the Bourne Shell. Now it's compatible with many others including BASH, Korn & C.
What is it?	Linux is an example of Open Source software development and Free Operating System (OS).	Unix is an operating system that is very popular in universities, companies, big enterprises etc.
GUI	Linux typically provides two GUIs, KDE and Gnome. But there are millions of alternatives such as LXDE, Xfce, Unity, Mate, twm, ect.	Initially Unix was a command based OS, but later a GUI was created called Common Desktop Environment. Most distributions now ship with Gnome.
Price	Free but support is available for a price.	Some free for development use (Solaris) but support is available for a price.

	Linux	Unix
Security	Linux has had about 60-100 viruses listed till date. None of them actively spreading nowadays.	A rough estimate of UNIX viruses is between 85 -120 viruses reported till date.
Threat detection and solution	In case of Linux, threat detection and solution is very fast, as Linux is mainly community driven and whenever any Linux user posts any kind of threat, several developers start working on it from different parts of the world	Because of the proprietary nature of the original Unix, users have to wait for a while, to get the proper bug fixing patch. But these are not as common.
Processors	Dozens of different kinds.	x86/x64, Sparc, Power, Itanium, PA-RISC, PowerPC and many others.
Examples	Ubuntu, Fedora, Red Hat, Debian, Archlinux, Android etc.	OS X, Solaris, All Linux
Architectures	Originally developed for Intel's x86 hardware, ports available for over two dozen CPU types including ARM	is available on PA-RISC and Itanium machines. Solaris also available for x86/x64 based systems.OSX is PowerPC(10.0-10.5)/x86(10.4)/x64(10.5-10.8)
Inception	Inspired by MINIX (a Unix-like system) and eventually after adding many features of GUI, Drivers etc, Linus Torvalds developed the framework of the OS that became LINUX in 1992. The LINUX kernel was released on 17th September, 1991	In 1969, it was developed by a group of AT&T employees at Bell Labs and Dennis Ritchie. It was written in “C” language and was designed to be a portable, multi-tasking and multi-user system in a time-sharing configuration.

Features of UNIX

High reliability, scalability and powerful features make UNIX a popular operating system, according to Intel. Now beyond its 40th year as of 2010, UNIX is the backbone of many data centers including the Internet. Big players using UNIX include Sun Microsystems, Apple Inc., Hewlett-Packard and AT&T, which is the original parent company of UNIX. The Open Group owns all UNIX specifications and the trademark, which are freely accessible and available over the Internet.

Multitasking and Portability

The main features of UNIX include multiuser, multitasking and portability capabilities. Multiple users access the system by connecting to points known as terminals. Several users can run multiple programs or processes simultaneously on one system. UNIX uses a high-level language that is easy to comprehend, modify and transfer to other machines, which means you can change language codes according to the requirements of new hardware on your computer. You, therefore, have the flexibility to choose any hardware, modify the UNIX codes accordingly and use UNIX across multiple architectures.

The Kernel and the Shell

The hub of a UNIX operating system, the kernel manages the applications and peripherals on a system. Together, the kernel and the shell carry out your requests and commands. You communicate with your system through the UNIX shell, which translates to the kernel. When you turn on your terminal, a system process starts that overlooks your inputs. When you enter your password, the system associates the shell program with your terminal. The shell allows you to customize options even if you are not technically savvy. For example, if you partially type a command, the shell anticipates the command for which you are aiming and displays the command for you. The UNIX shell is a program that gives and displays your prompts and, in conjunction with the kernel, executes your commands. The shell even maintains a history of the commands you enter, allowing you to reuse a command by scrolling through your history of commands.

Files and Processes

All the functions in UNIX involve either a file or a process. Processes are executions of programs, while files are collections of data created by you. Files may include a document, programming instructions for the system or a directory. UNIX uses a hierarchical file structure in its design that starts with a root directory--signified by the forward slash (/). The root is followed by its subdirectories, as in an inverted tree, and ends with the file. In the example "/Demand/Articles/UNIX.doc," the main directory "Demand" has a subdirectory "Articles," which has a file "UNIX.doc."

Features of Linux

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

- **Portable** – Portability means softwares can works on different types of hardwares in same way.Linux kernel and application programs supports 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 system resources like memory/ ram/ application programs at same time.
- **Multiprogramming** – Linux is a multiprogramming system means multiple applications can 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, call application programs etc.
- **Security** – Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

ARCHITECTURE OF UNIX

There are two important divisions in UNIX operating system architecture.
1.Kernel
2.Shell
In simple words you can say –

- Kernal – interacts with the machine's hardware
- Shell – interacts with the user

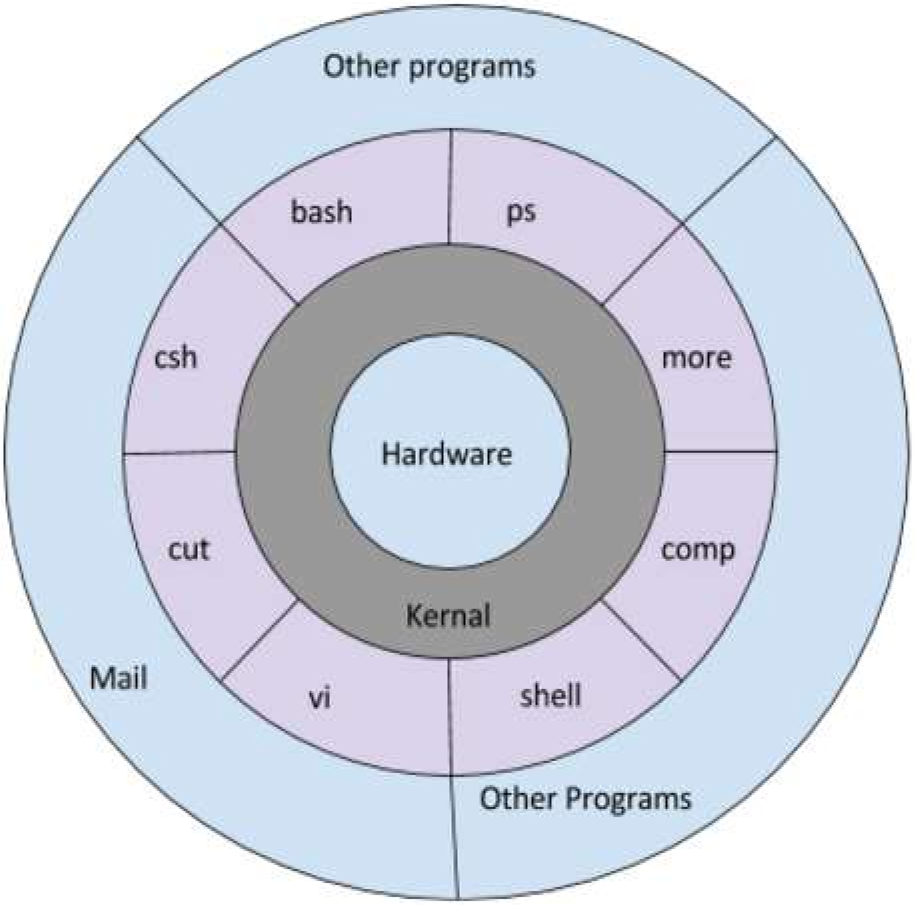
THE KERNEL:

The kernel of UNIX is the hub (or core) of the UNIX operating system. Kernel is a set of routines mostly written in C language.

User programs that need to access the hardware (like hard disk or terminal) use the services of the Kernel, which performs the job on the user's behalf.

User interacts with the Kernal by using System calls. Kernel allocates memory and time to programs and handles the file store and communications in response to system calls.

As an illustration of the way that the unix shell and the kernel work together, suppose a user types *mv myfile myfile1* (which has the effect of renaming the file myfile). The unix shell searches the file store for the file containing the program mv, and then requests the kernel, through system calls, to execute the program *mv* on *myfile*. When the process *mv myfile* has finished running, the unix shell then returns the UNIX prompt to the user, indicating that it is waiting for further commands.



THE SHELL:

UNIX Shell acts as a medium between the user and the kernel in unix system. When a user logs in, the login program checks the username and password and then starts another program called the shell.

Computers don't have any inherent capability of translating commands into action. This requires a command line interpreter (CLI) and this is handled by the "Outer Part" of the operating system i.e. Shell. It interprets the commands the user types in and arranges for them to be carried out.

In every unix system, the user can customize his own shell, and users can use different shells on the same machine.

The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the cursor keys to scroll up and down the list or type history for a list of previous commands.

Various Distributions of UNIX available in the market

Flavors that are available commercially (read: sold) include:
Solaris – Sun Microsystems' implementation, of which there are different kinds available: these are Solaris OS for SPARC platforms, Solaris OS for x86 platforms, and Trusted Solaris for both SPARC & x86 platforms; the latest version is Solaris 10 OS
AIX – short for Advanced Interactive eXecutive; IBM's implementation, the latest release of which, is the AIX 5L version 5.2.
SCO UnixWare and OpenServer – are implementations derived from the original AT&T Unix® source code acquired by the Santa Cruz Operation Inc. from Novell, and later on bought by Caldera Systems; the latest versions are UnixWare 7.1.3 and OpenServer 5.0.7
BSD/OS – the Berkeley Software Distribution (BSD) Unix implementation from Wind River; its latest version is the BSD/OS 5.1 Internet Server Edition
IRIX – the proprietary version of Unix from Silicon Graphics Inc.; the latest release of which is IRIX 6.5
HP-UX – short for Hewlett-Packard UniX; the latest version is the HP-UX 11i
Tru64 UNIX – the Unix operating environment for HP AlphaServer systems; Tru64 UNIX v5.1B-1 is the latest version
Mac OS – Mac operating system from Apple Computer Inc. having a Unix core; the latest version is the Mac OS X Panther

Various Distributions of LINUX available in the market

Debian

Debian is one of the oldest distributions out there, and some newer, more popular distributions are based on the Debian software. For instance, Ubuntu is a newer Linux distribution based on the Debian architecture. Debian has more than 1,000 volunteers, which makes it very versatile.

Fedora and Red Hat

Red Hat was developed several years ago, and it was known as one of the easiest of the Linux distributions to learn, if you were new to the operating system. Fedora has a great support community, and it's known as one of the better business suites out there.

Ubuntu

For users who are unfamiliar with Linux but want to learn, Ubuntu is the closest to Windows and user-friendly you can get. Ubuntu is one of the babies of the Linux family. It's not that old, but its popularity has grown. Ubuntu has a very “Windows-like” interface, so it’s most popular for its ease of use for Windows users who want to migrate to a Linux platform.

Linux Mint

Mint has one of the coolest interfaces for customizing a desktop. Mint’s graphics and desktop reminds most old Linux users of the classic GNOME interface. Mint is the alternative to Ubuntu. Most Ubuntu fans who are unhappy with the way Ubuntu is heading moved to Mint.

Installation of UNIX

Use this procedure as the starting point for installing WebLogic Integration - Business Connect on the supported UNIX operating systems.

Steps

1. Create a user account for WebLogic Integration - Business Connect (connect, for example) as the home directory for the application. For example, you can use one of the following as a home directory:

   /opt/connect

   /usr/local/connect

2. Determine the device name of your CD-ROM drive.

   The installation CD has a standard ISO-9660 (High Sierra) file system with Rock Ridge extensions.

3. Determine how much RAM your server has. You need to enter this information during the installation routine.

4. See the installation procedure for your operating system:

- Installing on Hewlett-Packard HP-UX
- Installing on IBM AIX
- Installing on Sun Solaris

We can use the install.sh command:

Booting of UNIX

Unix boot process has these main phases:

- Basic hardware detection (memory, disk, keyboard, mouse, and the like).
- Executing the firmware system initialization program (happens automatically).
- Locating and running the initial boot program (by the firmware boot program), usually from a predetermined location on disk. This program may perform additional hardware checks prior to loading the kernel.
- Locating and starting the Unix kernel (by the first-stage boot program). The kernel image file to execute may be determined automatically or via input to the boot program.
- The kernel initializes itself and then performs final, high-level hardware checks, loading device drivers and/or kernel modules as required.
- The kernel starts the init process, which in turn starts system processes (daemons) and initializes all active subsystems. When everything is ready, the system begins accepting user logins.

Shutting Down of Unix System

From time to time, you will need to shut thesystem down. This is necessary for scheduled maintenance, running diagnostics, hardware changes or additions, and other administrative tasks.

During a clean system shutdown, the following actions take place:

- All users are notified that the system will be going down, preferably giving them some reasonable advance warning.
- All running processes are sent a signal telling them to terminate, allowing them time to exit gracefully, provided the program has made provisions to do so.
- All subsystems are shut down gracefully, via the commands they provide for doing so.
- All remaining users are logged off, and remaining processes are killed.
- Filesystem integrity is maintained by completing all pending disk updates.
- Depending on the type of shutdown, the system moves to single-user mode, the processor is halted, or the system is rebooted.

Installation of Linux

1. Download the **Linux** distribution of your choice. ...
2. Boot into the Live CD or Live USB.
3. Try out the **Linux** distribution before **installing**.
4. Start the **installation** process.

5. Create a username and password.

6. **Set up the partition.** Linux needs to be installed on a separate partition from any other operating systems on your computer if you intend dual booting Linux with another OS. A partition is a portion of the hard drive that is formatted specifically for that operating system.

7. **Boot into Linux.** Once the installation is finished, your computer will reboot. You will see a new screen when your computer boots up called “GNU GRUB”. This is a boot loader that handles Linux installations.Check your hardware.

8.Check your hardware. Most hardware should work out of the box with your Linux distro, though you may need to download some additional drivers to get everything working. Some hardware requires proprietary drivers to work correctly in Linux.

Booting of Linux

1.BIOS
- BIOS stands for Basic Input/Output System
- Performs some system integrity checks

2. MBR
- MBR stands for Master Boot Record.
- It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda

3. GRUB
- GRUB stands for Grand Unified Bootloader.
- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- As you notice from the above info, it contains kernel and initrd image.
- So, in simple terms GRUB just loads and executes Kernel and initrd images.

4. Kernel
- Mounts the root file system as specified in the “root=” in grub.conf
- Kernel executes the /sbin/init program

5. Init
- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
  - 0 – halt
  - 1 – Single user mode
- Typically you would set the default run level to either 3 or 5.

6. Runlevel programs
- When the Linux system is booting up, you might see various services getting started. For example, it might say “starting sendmail …. OK”. Those are the runlevel programs, executed from the run level directory as defined by your run level.

Shutdown of Linux

Linux was not made to be shut down, but if you really must, use the **shutdown** command. After completing the shutdown procedure, the -h option will halt the system, while -r will reboot it. The **reboot** and **halt** commands are now able to invoke **shutdown** if run when the system is in run levels 1-5, and thus ensure proper shutdown of the system, but it is a bad habit to get into, as not all UNIX/Linux versions have this feature. If your computer does not power itself down, you should not turn off the computer until you see a message indicating that the system is halted or finished shutting down, in order to give the system the time to unmount all partitions. Being impatient may cause data loss.

POSSIBLE QUESTIONS
PART B QUESTIONS
(EACH QUESTION CARRIES TWO MARKS)

1. What are the features of Unix ?
2. What are the features of Linux?
3. What do you mean by shell in Linux?
4. What do you mean by kernel in Unix?.
5. List out the various distributions of Unix.
6. List out the various distributions of Linux.
7. What is booting in Unix?
8. What is utility in Unix?.

PART C QUESTIONS
(EACH QUESTION CARRIES 6 MARKS)

1.Explain the difference between Unix and Linux.
2.Discuss the architecture of Unix.
3.Explain the architecture of Linux.
4.Discuss the various distributions of Unix and Linux.
5.Explain the internal and external commands in Linux.
6.Explain the Installation, Booting and shutdown process of Unix.

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - I : (Objective Type Multiple choice Questions each Question carries one Mark)

UNIX/LINUX PROGRAMMING

PART - A (Online Examination)

Questions	Opt1	opt2	opt3	opt4	KEY
computer hardware and acts as an intermediary	acceleration	Operating Syster	compiler	logical transcation	System
_____manages the execution of user programs to	resource all	work station	main frame	control program	program
____were the first computers used to tackle many	Mainframe computer system	Mainframe computer service	multiframe computer system	multiframe computer service	Mainframe computer system
_______________contains the address of an instruction to be fetched from memory	Program cou	Instruction regist	Control registers	Status registers	Instruction register (IR)
_______________ is also known as parallel systems	Multiprocessor systems	desktop systems	Time sharing systems	Multiprogrammed systems	Multiprocessor systems
______________operating systems are even more co	Time-sharin	desktop systems	Multiprogrammed systems	Multiprocessor systems	Time-sharing
__________ operating system keeps several jobs in memory simultaneously.	Time-sharin	desktop systems	Multiprogrammed systems	Multiprocessor systems	Multiprogrammed systems
___________can save more money than multiple sin	Multiprocessor systems	desktop systems	Time sharing systems	Multiprogrammed systems	Multiprocessor systems
The most common multiple-processor systems now u	symmetric multiprocessing	asymmetric multiprocessing	multithreading	multiprogramming	symmetric multiprocessing

Another form of a special-purpose operating system i	real-time system	distributed operating system	Process states	multiframe computer system	real-time system
The assignment of the CPU to the first process on the	graceful degradation	Time-sharing	dispatching	Multiprocessor systems	dispatching
The manifestation of a process in an operating system	Process state transitions	process control block	child process	cooperating processes	process control block
For multiprogramming operating system	special support from processor is essential	special support from processor is not essential	cache memory is essential	cache memory is not essential	special support from processor is not essential
Which operating system reacts in the actual time	Batch system	Quick response system	Real time system	Time sharing system	Real time system
The primary job of an OS is to ________	command resource	manage resource	provide utilities	Be user friendly	manage resource
The term " Operating System " means ________	A set of programs which controls computer working	The way a computer operator works	Conversion of high-level language in to machine level language	The way a floppy disk drive operates	A set of programs which controls computer working
With .... more than one process can be running simultaneously each on a different processer.	Multiprogramming	Uniprocessing	Multiprocessing	Uniprogramming	Multiprogramming

The two central themes of modern operating system are	Multiprogramming and Distributed processing	Multiprogramming and Central Processing	Single Programming and Distributed processing	None of above	Multiprogramming and Distributed processing
……………….. is a example of an operating system that support single user process and single thread	UNIX	MS-DOS	OS/2	Windows 2000	MS-DOS
The operating system of a computer serves as a software interface between the user and the ________.	Hardware	Peripheral	Memory	Screen	Hardware
What is a shell	It is a hardware component	It is a command interpreter	It is a part in compiler	It is a tool in CPU scheduling	It is a command interpreter
The main function of the command interpreter is:	to get and execute the next user-specified command	to provide the interface between the API and application program	to handle the files in operating system	none of the mentioned	to get and execute the next user-specified command
As OS that has strict time constraints	Sensor Node OS	Real Time OS	Mainframe OS	Timesharing OS	Real Time OS
The OS that groups similar jobs is called as	Network OS	Distributed OS	Mainframe OS	Batch OS	Batch OS
_____ systems are required to complete a critical task within a guaranteed amount of time.	hard real time	Priority inversion	load sharing	Priority inheritance	hard real time
A system program that combines the separately compiled modules of a program into a form suitable for execution	assembler	linking loader	cross compiler	load and go	linking loader
A _______________manages the execution of user programs to prevent errors and improper use of the computer.	Control program	Managing Program	allocating program	User program	Control program

________ is a program associated with the operating system but are not part of the kernel,	System Program	User program	System calls	Functions	System Program
General-purpose computers run most of their programs from rewriteable memory, called as _________________	Floppy disk	ROM	Random access Memory	Hard disk	Random access Memory
On systems with multiple command interpreters to choose from, the interpreters are known as _________	GUI	shells	Signal	Command	shells
The term PDA is ______________	Personal Digital Assistant	Personal Data Assistant	Personal Data Accountant	Private Digital Assistant	Personal Digital Assistant
_______________ handle large numbers of small requests	Batch systems	Time sharing	Transaction-processing systems	Distributed systems	Transaction-processing systems
The occurrence of an event is usually signaled by an ___________from either the hardware or the software.	interrupt	signal	service	routine	interrupt
Operating systems have a ______________for each device controller	Process	device driver	controller	allocator	device driver
CPU design that includes multiple computing cores on a single chip. Such multiprocessor systems are termed __________	multicore	uniprocessor	singlecore	multichips	multicore
logical storage unit is called as ___________	folder	file	RAM	ROM	file
___________ is any mechanism for controlling the access of processes or users to the resources defined by a computer system.	Protection	authorization	policy	privacy	Protection
A _____________is an operating system that provides features such as file sharing across the network.	network operating system	Distributed OS	Parallel OS	Sensor OS	network operating system
______________operating systems are even more complex than multi programmed operating systems.	Time-sharing	desktop systems	Multiprogrammed systems	Multiprocessor systems	Time-sharing

___________can save more money than multiple single-processor systems	Multiprocessor systems	desktop systems	Time sharing systems	Multiprogrammed systems	Multiprocessor systems
_______________ is also known as parallel systems or tightly coupled systems	Multiprocessor systems	desktop systems	Time sharing systems	Multiprogrammed systems	Multiprocessor systems
Another form of a special-purpose operating system is the	real-time system	distributed operating system	Process states	multiframe computer system	real-time system
The message-passing facility in Windows 2000 is called	MUTUAL EXCLUSION	Buffering	local procedure call facility	CRITICAL SECTIONS	local procedure call facility
Which process is known for initializing a microcomputer with its OS	cold booting	boot recording	booting	warm booting	booting
A series of statements explaining how the data is to be processed is called	instruction	compiler	program	interpretor	program
Distributed systems should	high security	have better resource sharing	better system utilization	low system overhead	have better resource sharing
Which of the following is always there in a computer	Batch system	Operating system	Time sharing system	Controlling system	Operating system
When did IBM released the first version of its disk operating system DOS version 1.0	1981	1982	1983	1984	1981
The kernel is a___________________	memory manager	resource manager	file manager	directory manager	resource manager
_______________contains the address of an instruction to be fetched from memory	Program counter (PC)	Instruction register (IR)	Control registers	Status registers	Instruction register (IR)
___________________contains the instruction most recently fetched.	Program counter (PC)	Instruction register (IR)	Control registers	Status registers	Program counter (PC)
If a process fails, most operating system write the error information to a	log file	another running process	new file	none of the mentioned	log file

The OS X has ________	monolithic kernel	hybrid kernel	microkernel	monolithic kernel with modules	hybrid kernel
Which Operating system does not support long file names	OS/2	Windows 95	MS-DOS	Windows NT	MS-DOS
Which Operating system does not support networking between computers	Windows 3.1	Windows 95	Windows 2000	Windows NT	Windows 3.1
Which Operating system is better for implementing client server network	MS DOS	Windows 95	Windows 98	Windows 2000	Windows 2000
___________is the commercial UNIX-based operating system of Sun Microsystems.	Solaris	UNIX	Linux	Macintosh	Solaris
__________ is an example of an open-source operating system	GNU/Linux	Windows 3.1	Windows NT	Macintosh	GNU/Linux
In Operating System__________ hides the details of how	Resource manager	Resource Abstraction	Resource Hiding	Information Hiding	Resource Abstraction

UNIT – II

SYLLABUS

System processes (an overview), External and internal commands, Creation of partitions in OS, Processes and its creation phases – Fork, Exec, wait

• Mode bit: Supervisor or User mode • Supervisor mode – Can execute all machine instructions – Can reference all memory locations • User mode – Can only execute a subset of instructions – Can only reference a subset of memory locations

Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

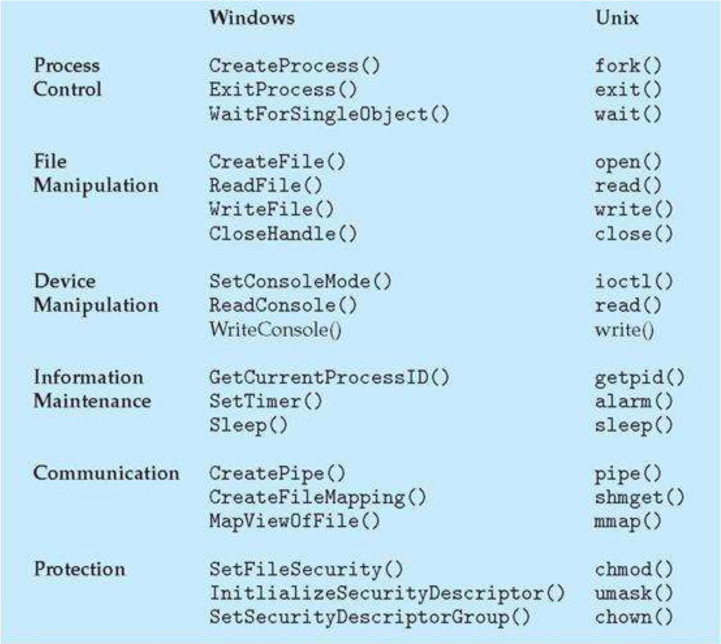
User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

SYSTEM CALLS AND SYSTEM PROGRAMS

- System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do. In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process. It is because of the critical nature of operations that the operating system itself does them every time they are needed. For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

- System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls.



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

SYSTEM CALL

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

In a typical UNIX system, there are around 300 system calls. Some of them which are important ones in this context are described below.

SYSTEM PROGRAMS

These programs are not usually part of the OS kernel, but are part of the overall operating system.

File Management

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

Status Information

Some programs simply request the date and time, and other simple requests. Others provide detailed performance, logging, and debugging information. The output of these files is often sent to a terminal window or GUI window

File modification

Programs such as text editors are used to create, and modify files.

Communications

These programs provide the mechanism for creating a virtual connect among processes, users, and other computers. Email and web browsers are a couple examples.

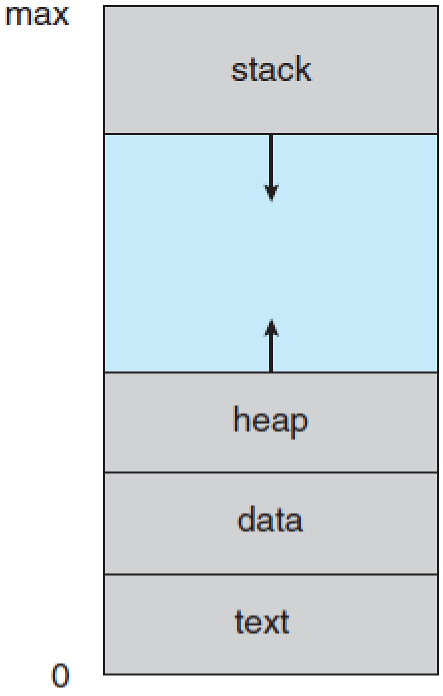
PROCESS MANAGEMENT

Process Concept

Process is a program that is in execution. It is defined as unit of work in modern systems. A batch system executes jobs, whereas a time-shared system has user programs, or tasks. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package. And even if a user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes.

Process in memory

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



Process in Memory

A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Process State

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- New. The process is being created.

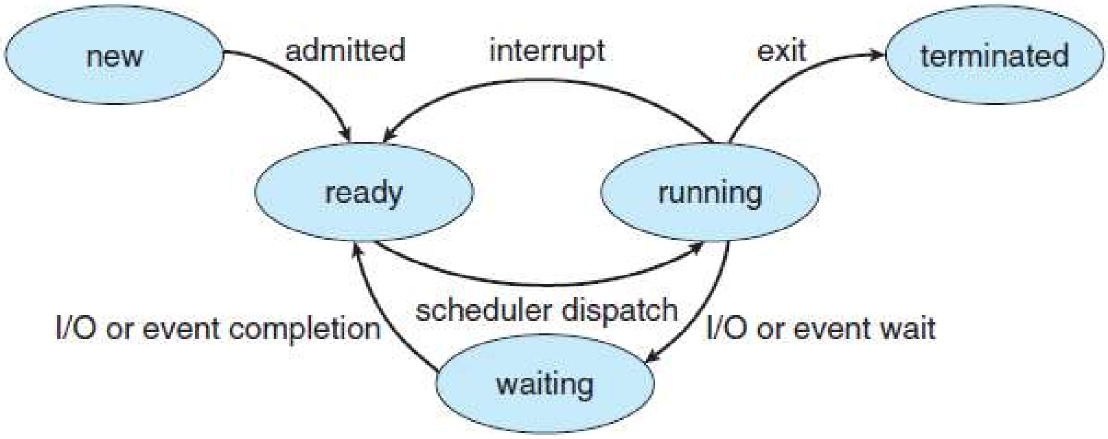
• Running. Instructions are being executed.

• Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• Ready. The process is waiting to be assigned to a processor.

• Terminated. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in the following Figure.



Process State Diagram

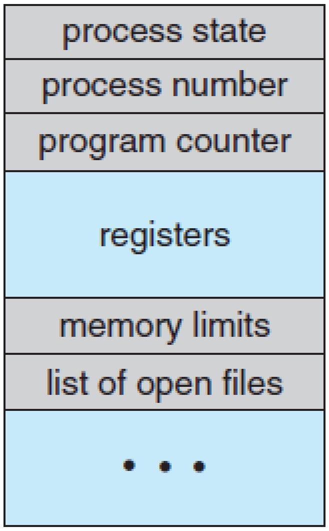
Process Control Block (PCB)

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. It contains many pieces of information associated with a specific process, including these: Process state. The state may be new, ready, running, and waiting, halted, and so on.

- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-

purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

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



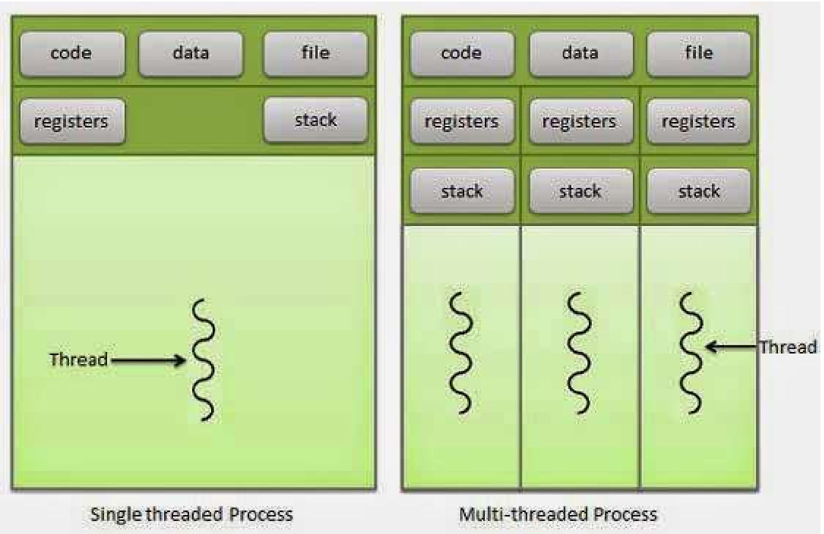
Process Control Block (PCB)

- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

THREAD

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

- Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. Following figure shows the working of the single and multithreaded processes.



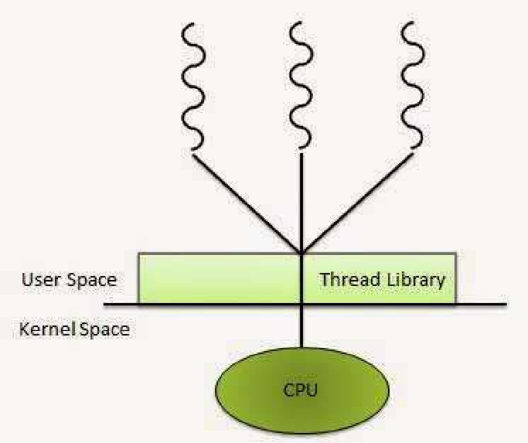
Advantages of Thread

- Thread minimizes context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- Economy- It is more economical to create and context switch threads.
- Utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

- Threads are implemented in following two ways
- User Level Threads -- User managed threads
- Kernel Level Threads -- Operating System managed threads acting on kernel, an operating system core.
- User Level Threads

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



ADVANTAGES

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

DISADVANTAGES

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

Kernel Level Threads

In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

ADVANTAGES

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

DISADVANTAGES

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

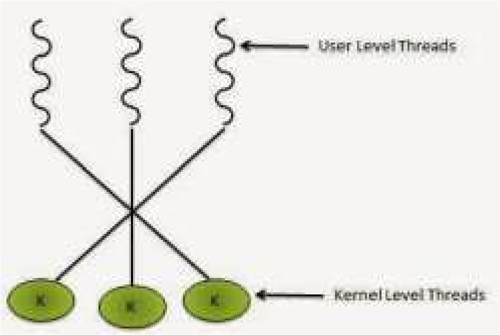
Multithreading Models

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

- Many too many relationships.
- Many to one relationship.
- One to one relationship.

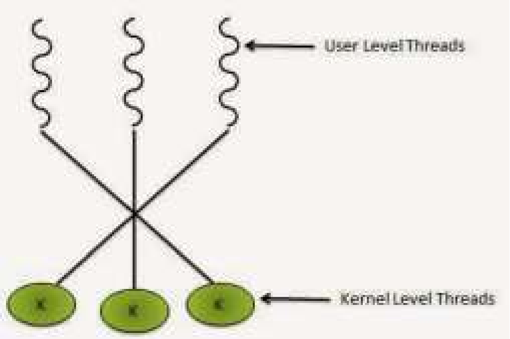
Many to Many Model

- In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.
- Following diagram shows the many to many models. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



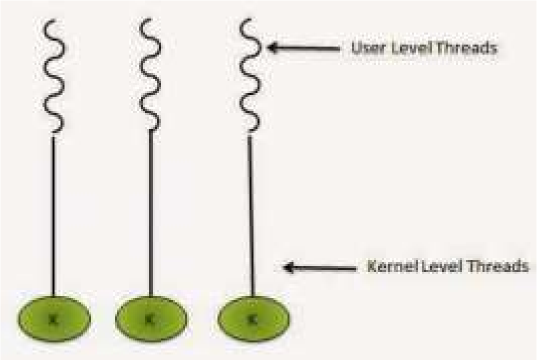
Many to One Model

- Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time,so multiple threads are unable to run in parallel on multiprocessors.
- If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.



One to One Model

- There is one to one relationship of user level thread to the kernel level thread. This model provides more concurrency than the many to one model. It is also another thread to run when a thread makes a blocking system call. It supports multiple thread to execute in parallel on microprocessors.
- Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



THREADING ISSUES OPERATING SYSTEMS

1. The fork() and exec() System Calls

- If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
- If a thread invokes the exec() system call, the program specified in the parameter to exec () will replace the entire process-including all threads.

2. Cancellation

- Thread cancellation is the task of terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- A thread that is to be canceled is often referred to as the **target thread**.
- Cancellation of a target thread may occur in two different scenarios:
  - **Asynchronous cancellation**. One thread immediately terminates the target thread.
  - **Deferred cancellation.** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
- The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.

3. Signal Handling

- A signal is used in UNIX systems to notify a process that a particular event has occurred. All signals, whether synchronous or asynchronous, follow the same pattern:
- A signal is generated by the occurrence of a particular event.
- A generated signal is delivered to a process.
- Once delivered, the signal must be handled.
- Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated.
- Every signal has a **default signal handler** that is run by the kernel when handling that signal. This default action can be overridden by a **user defined signal handler** that is called to handle the signal.
- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in

multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

- In general the following options exist:
  - Deliver the signal to the thread to which the signal applies.
  - Deliver the signal to every thread in the process.
  - Deliver the signal to certain threads in the process.
  - Assign a specific thread to receive all signals for the process.

4. Thread Pools

- The first **issue** concerns the amount of time required to create the thread prior to servicing the request, together with the fact that this thread will be discarded once it has completed its work.
- The second **issue** is more troublesome: if we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.
- The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work.
- Thread pools offer these benefits:

  - Servicing a request with an existing thread is usually faster than waiting to create a thread.
  - A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

5. Thread-Specific Data

- Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming.

- However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread specific data**.
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data.

PROCESS SCHEDULING

Definition

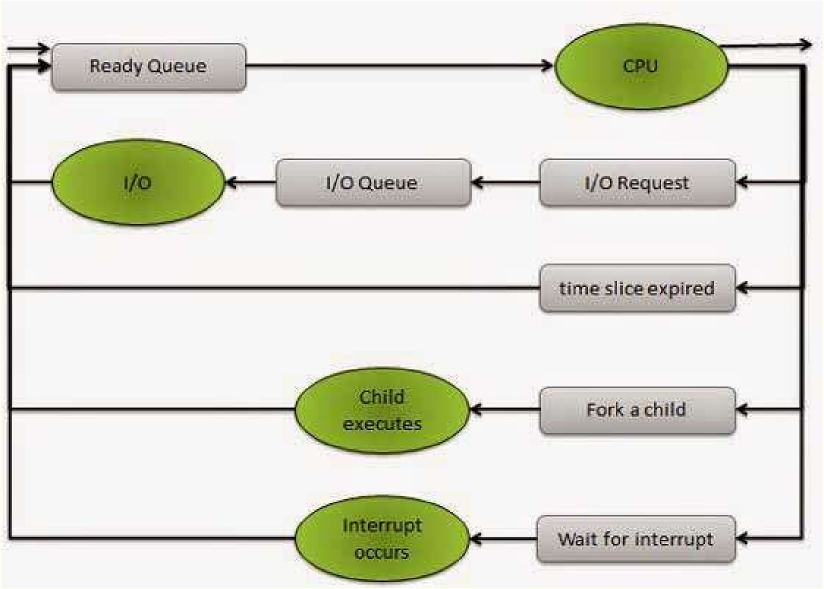
- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

Scheduling Queues

- Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

This figure shows the queuing diagram of process scheduling.

- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system.



Queues are of two types

- Ready queue
- Device queue

A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.

- The process could issue an I/O request and then it would be placed in an I/O queue.
- The process could create new sub process and will wait for its termination.
- The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

Two State Process Model

Two state process model refers to running and non-running states which are described below.

S.N.	State & Description
1	**Running** When new process is created by Operating System that process enters into the system as in the running state.
2	**Not Running** Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

Schedulers

Schedulers are special system software which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types

- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

Long Term Scheduler

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

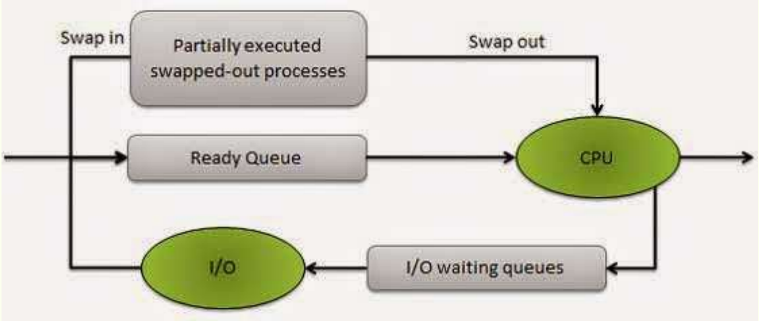
- On some systems, the long term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is use of long term scheduler.

Short Term Scheduler

- It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.
- Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

Medium Term Scheduler

- Medium term scheduling is part of the swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.



- Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison between Scheduler

S.N.	Long Term Scheduler	Short Term Scheduler	Medium Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

SCHEDULING (PREEMPTIVE AND NONPREEMPTIVE)

- *A* final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models.
- Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

General Goals

Fairness

Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement. Note that giving equivalent or equal time is not fair. Think of *safety control* and *payroll* at a nuclear plant.

Policy Enforcement

The scheduler has to make sure that system's policy is enforced. For example, if the local policy is safety then the *safety control processes* must be able to run whenever they want to, even if it means delay in *payroll processes*.

Efficiency

Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second than if some components are idle.

Response Time

A scheduler should minimize the response time for interactive user.

Turnaround

A scheduler should minimize the time batch users must wait for an output.

Throughput

A scheduler should maximize the number of jobs processed per unit time.

Preemptive Vs Non preemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Non preemptive Scheduling

- A scheduling discipline is non preemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Following are some characteristics of non preemptive scheduling

1. In non preemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In non preemptive system, response times are more predictable because incoming high priority jobs cannot displace waiting jobs.
3. In non preemptive scheduling, a scheduler executes jobs in the following two situations.
   a. When a process switches from running state to the waiting state.
   b. When a process terminates.

Preemptive Scheduling

- A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.
- The strategy of allowing processes that are logically runable to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

Schedule:

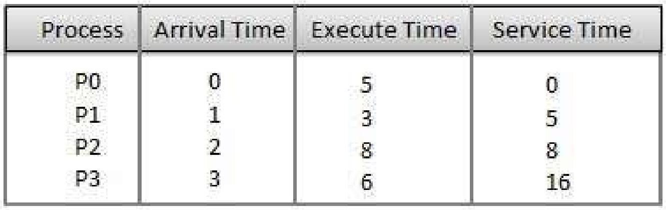
- A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

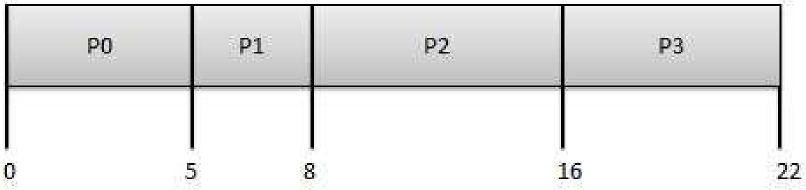
- These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state; it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows −

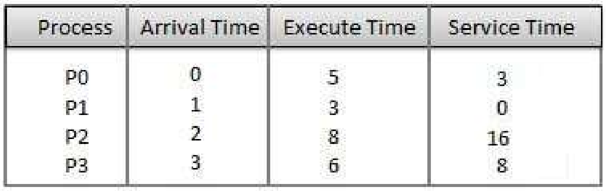
Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

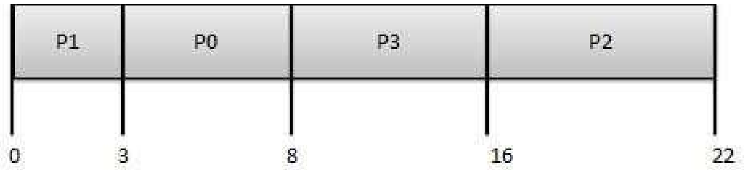
Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.

- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	3
P1	1	3	0
P2	2	8	16
P3	3	6	8



Wait time of each process is as follows −

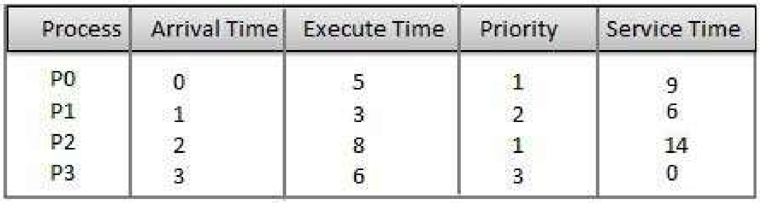
Process	Wait Time : Service Time - Arrival Time
P0	3 - 0 = 3
P1	0 - 0 = 0
P2	16 - 2 = 14
P3	8 - 3 = 5

Average Wait Time: (3+0+14+5) / 4 = 5.50

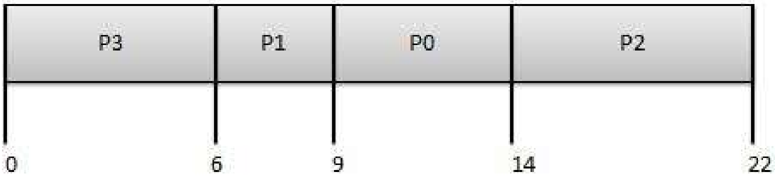
Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.

- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.



Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	9 - 0 = 9
P1	6 - 1 = 5
P2	14 - 2 = 12
P3	0 - 0 = 0

Average Wait Time: (9+5+12+0) / 4 = 6.5

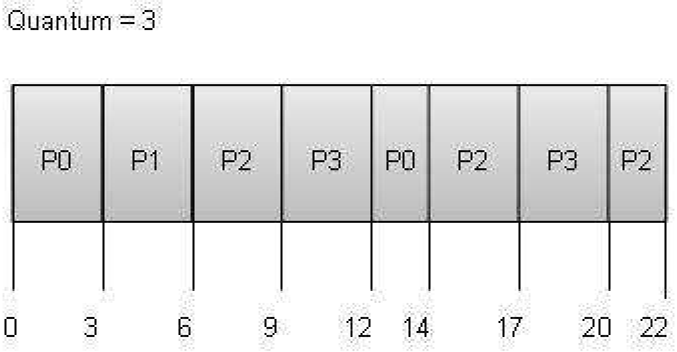
Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.

- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

Multiple-Level Queues Scheduling

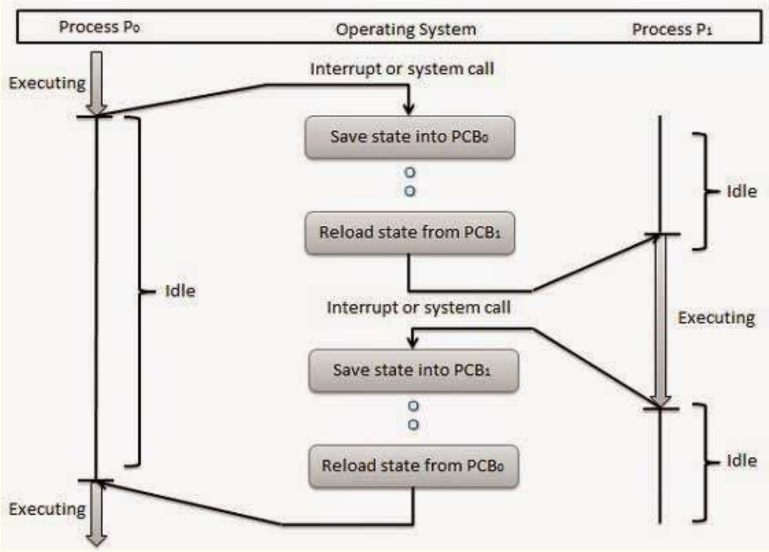
Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Context Switch

- A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.
- When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process.
- Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved. Content switching times are highly dependent on hardware support. Context switch requires (n + m) bxK time units to save the state of the processor with n general registers, assuming b are the store operations are required to save n and m registers of two process control blocks and each store instruction requires K time units.



Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time. When the process is switched, the following information is stored.

- Program Counter
- Scheduling Information
- Base and limit register value
- Currently used register
- Changed State
- I/O State
- Accounting

CONCURRENT AND PROCESSES

Co-operation of Process

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process. There are several reasons for providing an environment that allows process cooperation:

• **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file).It must provide an environment to allow concurrent access to such information.

• **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

• **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads,

• **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an inter-process communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of inter-process communication: shared memory and message passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

CRITICAL SECTION PROBLEM

Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

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

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

1. **Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

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

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

Two general approaches are used to handle critical sections in operating systems: preemptive kernels and non-preemptive kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

SEMAPHORE

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). The wait() operation was originally termed P (from the Dutch proberen, “to test”); signal() was originally called V (from verhogen, “to increment”). The definition of wait() is as follows:

```
wait(S) {
while (S <= 0)
; // busy wait
S--;
}
```

The definition of signal() is as follows:

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

All modifications to the integer value of the semaphore in the wait () and signal () operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

Semaphore Usage

Operating systems often distinguish between **counting and binary semaphores**. The value of a **counting semaphore** can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Thus, **binary semaphores** behave similarly to mutex locks. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal () operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Deadlock and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked. To illustrate this, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1: P0 P1

```
        wait(S); wait(Q);
        wait(Q); wait(S);
        . .
        . .
```

```
. .
signal(S); signal(Q);
signal(Q); signal(S);
```

Suppose that P0 executes wait(S) and then P1 executes wait(Q).When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal() operations cannot be executed, P0 and P1 are deadlocked. We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

The events with which we are mainly concerned here are resource acquisition and release Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

METHODS OF INTER-PROCESS COMMUNICATION (IPC)

Inter-process communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control.

The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-Passing Systems

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages. A message-passing facility provides at least two operations:

send(message) receive(message)

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system common kind of tradeoff seen throughout operating-system design. If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, but rather with its logical implementation. Here are several methods for logically implementing a link and the send()/receive() operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

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

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send () and receive () primitives are defined as follows:

• send (P, message)—Send a message to process P.

• receive (id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection.

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The send () and receive () primitives are defined as follows:

• send (A, message)—Send a message to mailbox A.

• receive (A, message)—Receive a message from mailbox A.

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

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

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

Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:

- Allow a link to be associated with two processes at most.
- Allow at most one process at a time to execute a receive () operation.
- Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message).

The system may define an algorithm for selecting which process will receive the message (for example, round robin, where processes take turns receiving messages). The system may identify the receiver to the sender. A mailbox may be owned either by a process or by the operating system.

If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears.

Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists. In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a mailbox.

The process that creates a new mailbox is that mailbox’s owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

Synchronization

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either blocking or non blocking— also known as synchronous and asynchronous. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

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

• Non-blocking send. The sending process sends the message and resumes operation.

• Blocking receive. The receiver blocks until a message is available.

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

Different combinations of send () and receive () are possible. When both send () and receive () are blocking, we have a rendezvous between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking send () and receive () statements. The producer merely invokes the blocking send () call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive (), it blocks until a message is available.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

• Zero capacity. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

• Bounded capacity. The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

• Unbounded capacity. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

DEADLOCK

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

1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. Release. The process releases the resource.

- For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure 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.
- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files).

Deadlock Prevention

Mutual Exclusion

➢ The mutual exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

- A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable. For example, a mutex lock cannot be simultaneously shared by several processes.

Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that we can use requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls. An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.
- Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. In the example given, for instance, we can release the DVD drive and disk file, and then request the disk file and printer, only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the beginning for both protocols.
- Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol.

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Circular Wait

- The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- To illustrate, we let R = {R1, R2, ..., Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function F: R→N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12

- We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type —say, Ri . After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri). For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.
- Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that F(Ri) ≥ F(Rj). Note also that if several instances of the same resource type are needed, a single request for all of them must be issued. If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be {P0, P1, ..., Pn}, where Pi is waiting for a resource Ri , which is held by process Pi+1. (Modulo arithmetic is used on the indexes, so that Pn is waiting for a resource Rn held by P0.) Then, since process Pi+1 is holding resource Ri while requesting resource Ri+1, we must have F(Ri) < F(Ri+1) for all i. But this condition means that F(R0) < F(R1) < ... < F(Rn) < F(R0). By transitivity, F(R0) < F(R0), which is impossible. Therefore, there can be no circular wait.

Deadlock Avoidance:

- For avoiding deadlocks, it is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive.
- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

- The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

Safe State:

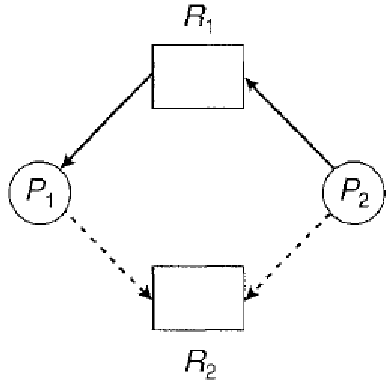
- A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.
- A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe. A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state.
-

Resource-Allocation-Graph Algorithm

- If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge.
- A claim edge Pi ~ Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process Pi requests resource R1, the claim edge P; -+ R1 is converted to a request edge. Similarly, when a resource R1 is released by P;, the assignment edge Rj -+ P; is reconverted to a claim edge P; -+ Rj.
- We note that the resources must be claimed a priori in the system. That is, before process P; starts executing, all its claim edges must already appear in the resource-allocation

graph. We can relax this condition by allowing a claim edge P; -+ R1 to be added to the graph only if all the edges associated with process P; are claim edges. Now suppose that process P; requests resource Rj. The request can be granted only if converting the request edge P; -+ Rj to an assignment edge R1 -+ P; does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm.

- An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of processes in the system. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process P; will have to wait for its requests to be satisfied. To illustrate this algorithm, we consider the following resource-allocation graph.



- Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph .A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

Banker’s algorithm:

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The Banker's algorithm is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This nun1.ber may not exceed the total number of resources in the system.

- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources. Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Ri are available.

Max: An n x m matrix defines the maximum demand of each process. If Max[i] [j] equals k, then process P; may request at most k instances of resource type Ri.

Allocation: An 11 x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals lc, then process P; is currently allocated lc instances of resource type Rj.

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process P; may need k more instances of resource type Ri to complete its task. Note that Need[i][j] equals Max[i][j] - Allocation [i][j].

These data structures vary over time in both size and value. To simplify the presentation of the banker's algorithm, we next establish some notation. Let X andY be vectors of length 11. We say that X::= Y if and only if X[i] ::= Y[i] for all i = 1, 2, ... , n. For example, if X = (1,7,3,2) and Y = (0,3,2,1), then Y ::=X. In addition, Y < X if Y ::=X and Y# X. We can treat each row in the matrices Allocation and Need as vectors and refer to them as Allocation; and Need;. The vector Allocation; specifies the resources currently allocated to process P;; the vector Need; specifies the additional resources that process P; may still request to complete its task.

<u>Safety Algorithm</u>

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work= Available and Finish[i] =false for i = 0, 1, ... , n - 1.

2. Find an index i such that both
    a. Finish[i] ==false
    b. Need$_i$ Work. If no such i exists, go to step 4.
3. Work = Work + Allocation; Finish[i] = true. Go to step 2.
4. If Finish[i] ==true for all i, then the system is in a safe state.

Resource-Request Algorithm

Let Request; be the request vector for process P;. If Request; [j] == k, then process P; wants k instances of resource type Rj. When a request for resources is made by process P;, the following actions are taken:

1. If Request; <= Need; go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request< = Available, go to step 3. Otherwise, P; must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P; by modifyil1.g the state as follows:

   Available= Available- Request$_i$

Deadlock

Allocation; =Allocation; +Request$_i$

Need; =Need$_i$- Request$_i$

If the resulting resource-allocation state is safe, the transaction is completed, and process P; is allocated its resources. However, if the new state is unsafe, then P; must wait for Request$_i$, and the old resource-allocation state is restored.

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

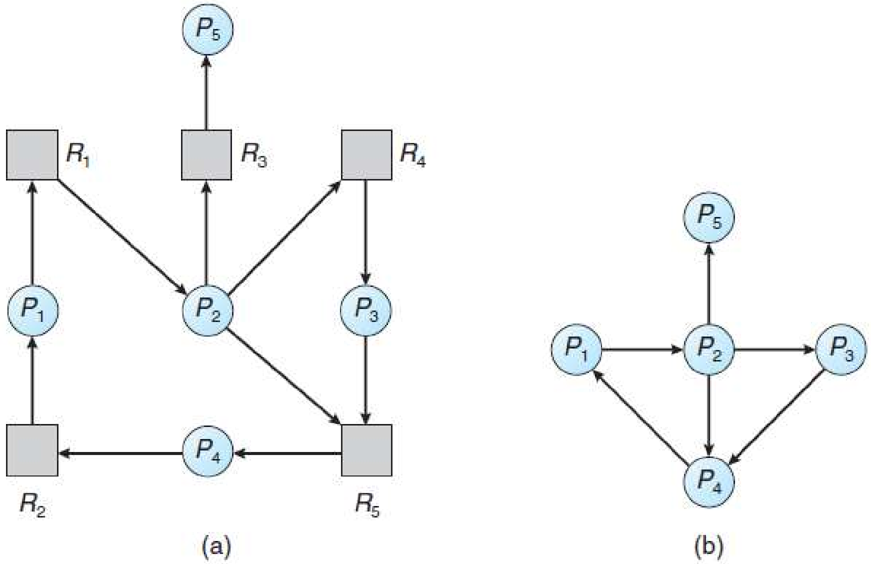
• An algorithm that examines the state of the system to determine whether a deadlock has occurred

• An algorithm to recover from the deadlock

Single Instance of Each Resource Type

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

More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.



An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq . In Figure, we present a resource-allocation graph and the corresponding wait-for graph. As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.

To detect deadlocks, the system needs to maintain the wait for graph and periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker’s algorithm

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

• Allocation. An n × m matrix defines the number of resources of each type currently allocated to each process.

• Request. An n × m matrix indicates the current request of each process. If Request[i][j] equals k, then process Pi is requesting k more instances of resource type Rj . To simplify notation, we again treat the rows in the matrices Allocation and Request as vectors; we refer to them as Allocationi and Requesti . The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available. For i = 0, 1, ..., n–1, if Allocationi _= 0, then Finish[i] = false. Otherwise, Finish[i] = true.

2. Find an index i such 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, 0≤i<n, then the system is in a deadlocked state. Moreover, if Finish[i] == false, then process Pi is deadlocked. This algorithm requires an order of m × n2 operations to detect whether the system is in a deadlocked state. You may wonder why we reclaim the resources of process Pi (in step 3) as soon as we determine that Requesti ≤ Work (in step 2b). We know that Pi is currently not involved in a deadlock (since Requesti ≤ Work). Thus, we take an optimistic attitude and assume that Pi will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T0, we have the following resource-allocation state:

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <P0, P2, P3, P1, P4> results in Finish[i] == true for all i. Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

	Request
	A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock.

Process Termination

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

- Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
2. Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as

necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

POSSIBLE QUESTIONS

UNIT – II

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. Define internal commands in Unix.
2. What is External commands in Linux.
3. What do you mean by partition in OS?
4. Define Fork command.
5. Define Wait commands

PART – C (6 MARKS)

1. Discuss in detail the System Processes.
2. Explain the external and internal commands.
3. Explain the creation of partitions in OS.
4. Explain the processes and its creation phases.
5. Discuss the commands of fork, exec and wait.

UNIT – II

SYLLABUS

System processes (an overview), External and internal commands, Creation of partitions in OS, Processes and its creation phases – Fork, Exec, wait

• Mode bit: Supervisor or User mode • Supervisor mode – Can execute all machine instructions – Can reference all memory locations • User mode – Can only execute a subset of instructions – Can only reference a subset of memory locations

Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

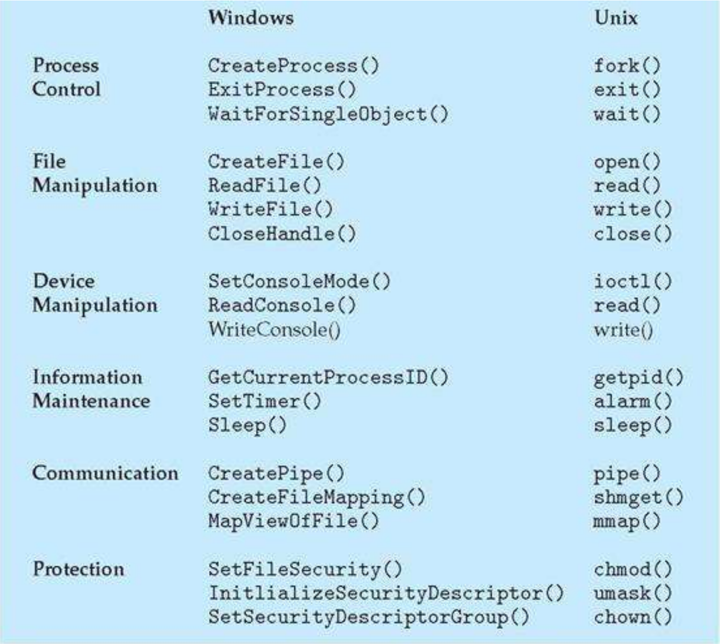
User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

SYSTEM CALLS AND SYSTEM PROGRAMS

- System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do. In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process. It is because of the critical nature of operations that the operating system itself does them every time they are needed. For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

- System programs provide basic functioning to users so that they do not need to write their own environment for program development (editors, compilers) and program execution (shells). In some sense, they are bundles of useful system calls.



	Windows	Unix

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

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

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

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

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

Protection	SetFileSecurity()
	InitlializeSecurityDescriptor()
	SetSecurityDescriptorGroup()
		chmod()
		umask()
		chown()

SYSTEM CALL

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

In a typical UNIX system, there are around 300 system calls. Some of them which are important ones in this context are described below.

SYSTEM PROGRAMS

These programs are not usually part of the OS kernel, but are part of the overall operating system.

File Management

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

Status Information

Some programs simply request the date and time, and other simple requests. Others provide detailed performance, logging, and debugging information. The output of these files is often sent to a terminal window or GUI window

File modification

Programs such as text editors are used to create, and modify files.

Communications

These programs provide the mechanism for creating a virtual connect among processes, users, and other computers. Email and web browsers are a couple examples.

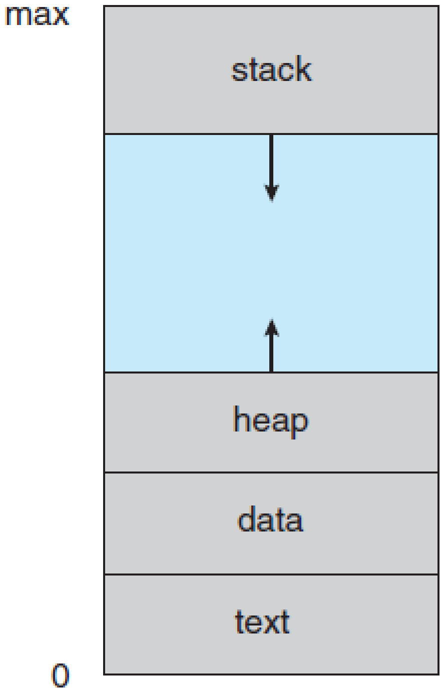
PROCESS MANAGEMENT

Process Concept

Process is a program that is in execution. It is defined as unit of work in modern systems. A batch system executes jobs, whereas a time-shared system has user programs, or tasks. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package. And even if a user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes.

Process in memory

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



Process in Memory

A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Process State

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- New. The process is being created.

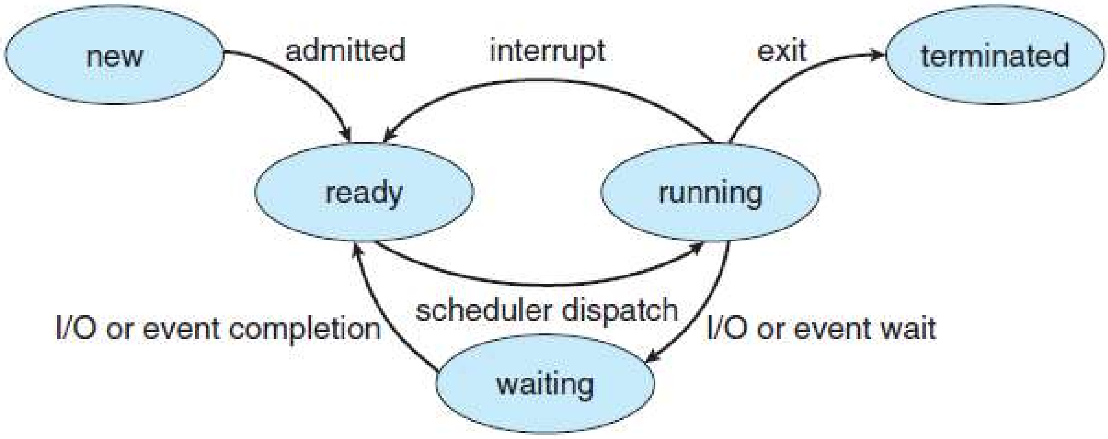
• Running. Instructions are being executed.

• Waiting. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• Ready. The process is waiting to be assigned to a processor.

• Terminated. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in the following Figure.



Process State Diagram

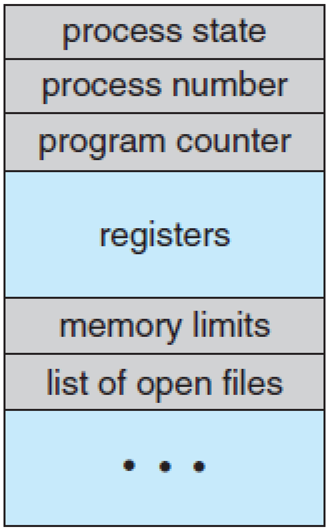
Process Control Block (PCB)

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. It contains many pieces of information associated with a specific process, including these: Process state. The state may be new, ready, running, and waiting, halted, and so on.

- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-

purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

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



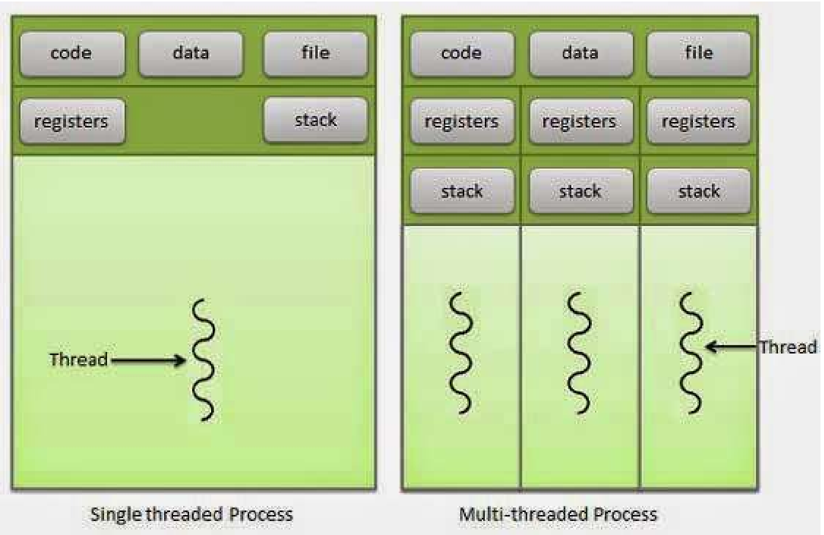
Process Control Block (PCB)

- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

THREAD

- A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

- Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. Following figure shows the working of the single and multithreaded processes.



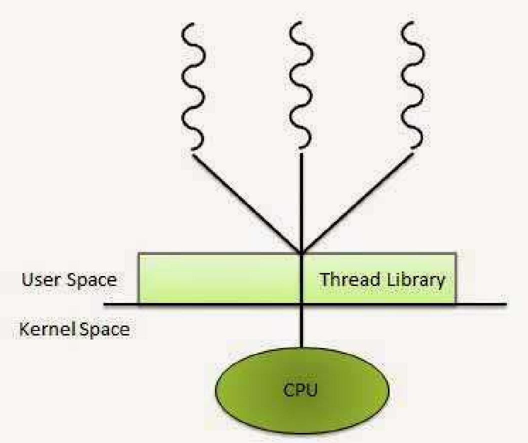
Advantages of Thread

- Thread minimizes context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- Economy- It is more economical to create and context switch threads.
- Utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

- Threads are implemented in following two ways
- User Level Threads -- User managed threads
- Kernel Level Threads -- Operating System managed threads acting on kernel, an operating system core.
- User Level Threads

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



ADVANTAGES

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

DISADVANTAGES

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

Kernel Level Threads

In this case, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

ADVANTAGES

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

DISADVANTAGES

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

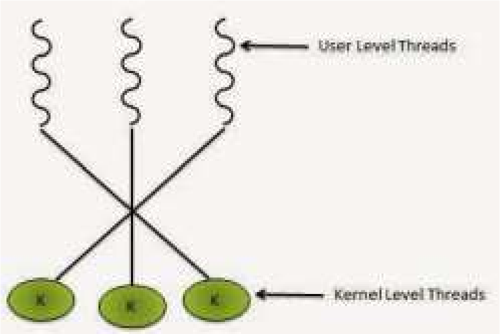
Multithreading Models

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

- Many too many relationships.
- Many to one relationship.
- One to one relationship.

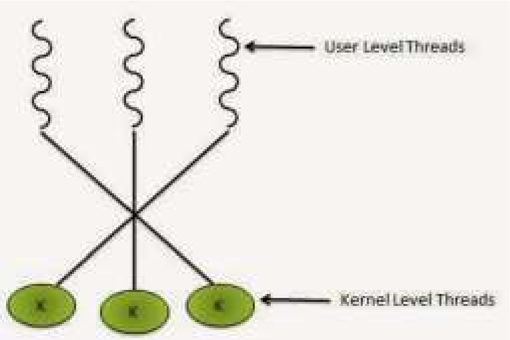
Many to Many Model

- In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.
- Following diagram shows the many to many models. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



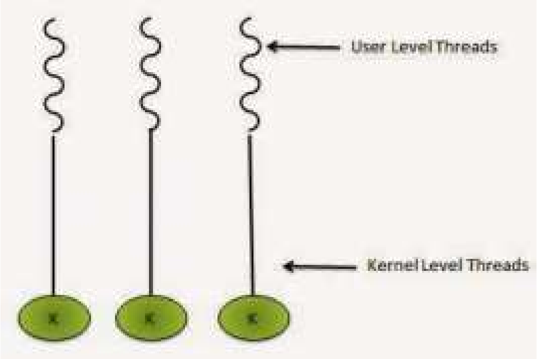
Many to One Model

- Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. Only one thread can access the Kernel at a time,so multiple threads are unable to run in parallel on multiprocessors.
- If the user level thread libraries are implemented in the operating system in such a way that system does not support them then Kernel threads use the many to one relationship modes.



One to One Model

- There is one to one relationship of user level thread to the kernel level thread. This model provides more concurrency than the many to one model. It is also another thread to run when a thread makes a blocking system call. It supports multiple thread to execute in parallel on microprocessors.
- Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



THREADING ISSUES OPERATING SYSTEMS

1. The fork() and exec() System Calls

- If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
- If a thread invokes the exec() system call, the program specified in the parameter to exec () will replace the entire process-including all threads.

2. Cancellation

- Thread cancellation is the task of terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- A thread that is to be canceled is often referred to as the **target thread**.
- Cancellation of a target thread may occur in two different scenarios:
  - **Asynchronous cancellation**. One thread immediately terminates the target thread.
  - **Deferred cancellation.** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.
- The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.

3. Signal Handling

- A signal is used in UNIX systems to notify a process that a particular event has occurred. All signals, whether synchronous or asynchronous, follow the same pattern:
- A signal is generated by the occurrence of a particular event.
- A generated signal is delivered to a process.
- Once delivered, the signal must be handled.
- Examples of synchronous signals include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated.
- Every signal has a **default signal handler** that is run by the kernel when handling that signal. This default action can be overridden by a **user defined signal handler** that is called to handle the signal.
- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in

multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

- In general the following options exist:
  - Deliver the signal to the thread to which the signal applies.
  - Deliver the signal to every thread in the process.
  - Deliver the signal to certain threads in the process.
  - Assign a specific thread to receive all signals for the process.

4. Thread Pools

- The first **issue** concerns the amount of time required to create the thread prior to servicing the request, together with the fact that this thread will be discarded once it has completed its work.
- The second **issue** is more troublesome: if we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.
- The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work.
- Thread pools offer these benefits:
  - Servicing a request with an existing thread is usually faster than waiting to create a thread.
  - A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

5. Thread-Specific Data

- Threads belonging to a process share the data of the process. Indeed, this sharing of data provides one of the benefits of multithreaded programming.

- However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread specific data**.
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data.

PROCESS SCHEDULING

Definition

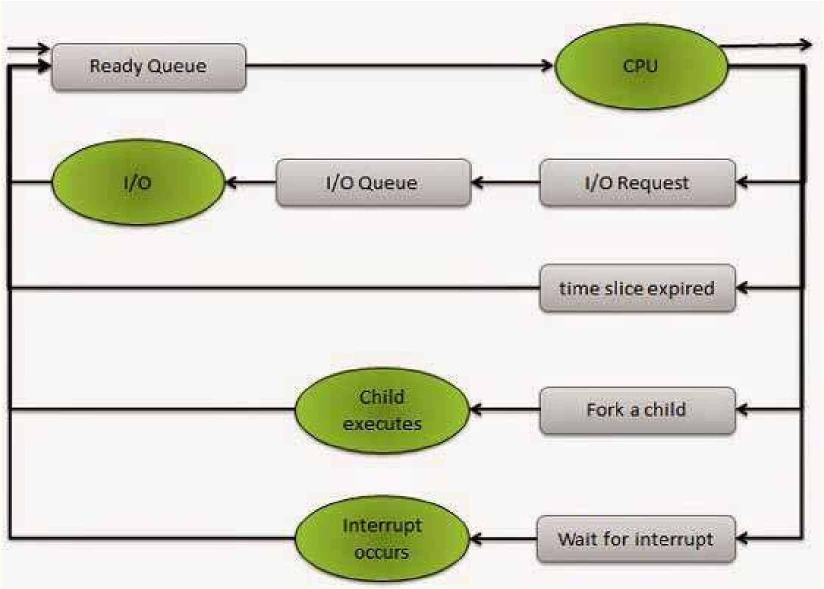
- ➢ The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- ➢ Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

Scheduling Queues

- ➢ Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a job queue. This queue consists of all processes in the system. The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

This figure shows the queuing diagram of process scheduling.

- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system.



Queues are of two types

- Ready queue
- Device queue

A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.

- The process could issue an I/O request and then it would be placed in an I/O queue.
- The process could create new sub process and will wait for its termination.
- The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

<u>Two State Process Model</u>

Two state process model refers to running and non-running states which are described below.

S.N.	State & Description
1	**Running** When new process is created by Operating System that process enters into the system as in the running state.
2	**Not Running** Processes that are not running are kept in queue, waiting for their turn to execute. Each entry in the queue is a pointer to a particular process. Queue is implemented by using linked list. Use of dispatcher is as follows. When a process is interrupted, that process is transferred in the waiting queue. If the process has completed or aborted, the process is discarded. In either case, the dispatcher then selects a process from the queue to execute.

Schedulers

Schedulers are special system software which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types

- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

Long Term Scheduler

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

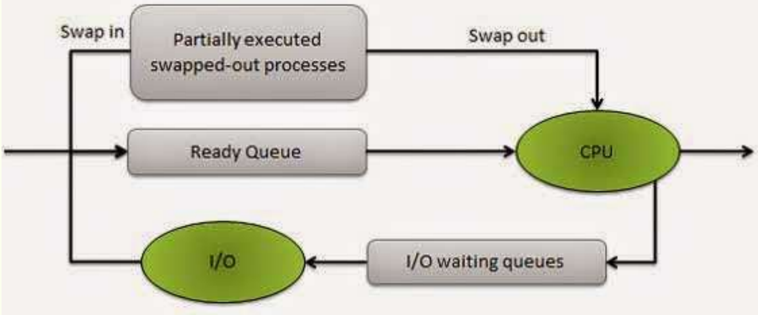
- On some systems, the long term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When process changes the state from new to ready, then there is use of long term scheduler.

Short Term Scheduler

- It is also called CPU scheduler. Main objective is increasing system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.
- Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

Medium Term Scheduler

- Medium term scheduling is part of the swapping. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.



- Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This process is called swapping, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

Comparison between Scheduler

S.N.	Long Term Scheduler	Short Term Scheduler	Medium Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

SCHEDULING (PREEMPTIVE AND NONPREEMPTIVE)

- *A* final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models.
- Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

General Goals

Fairness

Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement. Note that giving equivalent or equal time is not fair. Think of *safety control* and *payroll* at a nuclear plant.

Policy Enforcement

The scheduler has to make sure that system's policy is enforced. For example, if the local policy is safety then the *safety control processes* must be able to run whenever they want to, even if it means delay in *payroll processes*.

Efficiency

Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second than if some components are idle.

Response Time

A scheduler should minimize the response time for interactive user.

Turnaround

A scheduler should minimize the time batch users must wait for an output.

Throughput

A scheduler should maximize the number of jobs processed per unit time.

Preemptive Vs Non preemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Non preemptive Scheduling

- A scheduling discipline is non preemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Following are some characteristics of non preemptive scheduling

1. In non preemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In non preemptive system, response times are more predictable because incoming high priority jobs cannot displace waiting jobs.
3. In non preemptive scheduling, a scheduler executes jobs in the following two situations.
    a. When a process switches from running state to the waiting state.
    b. When a process terminates.

Preemptive Scheduling

- A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.
- The strategy of allowing processes that are logically runable to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

Schedule:

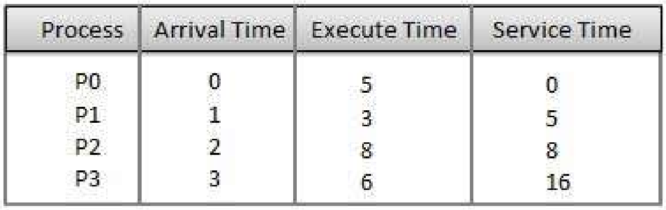
- A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter −

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

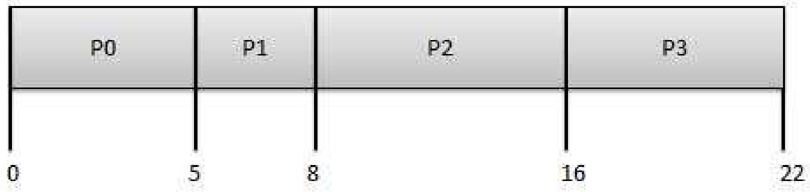
- These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state; it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows −

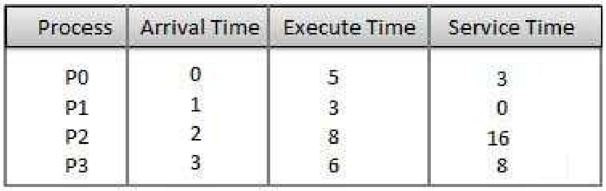
Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

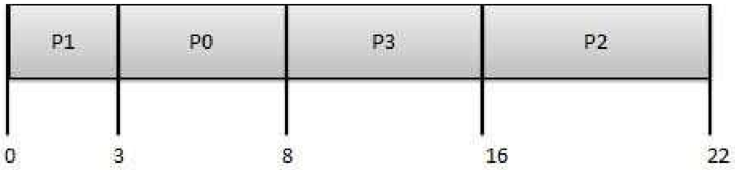
Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.

- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.



Process	Arrival Time	Execute Time	Service Time
P0	0	5	3
P1	1	3	0
P2	2	8	16
P3	3	6	8



Wait time of each process is as follows −

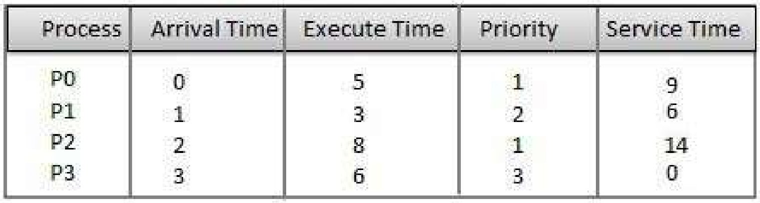
Process	Wait Time : Service Time - Arrival Time
P0	3 - 0 = 3
P1	0 - 0 = 0
P2	16 - 2 = 14
P3	8 - 3 = 5

Average Wait Time: (3+0+14+5) / 4 = 5.50

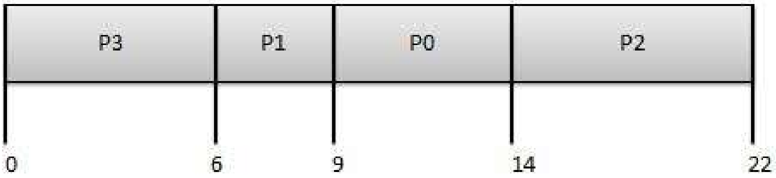
Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.

- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.



Process	Arrival Time	Execute Time	Priority	Service Time
P0	0	5	1	9
P1	1	3	2	6
P2	2	8	1	14
P3	3	6	3	0



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	9 - 0 = 9
P1	6 - 1 = 5
P2	14 - 2 = 12
P3	0 - 0 = 0

Average Wait Time: (9+5+12+0) / 4 = 6.5

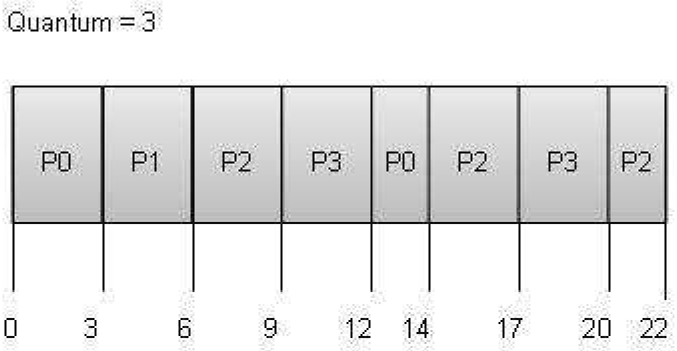
Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.

- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

Multiple-Level Queues Scheduling

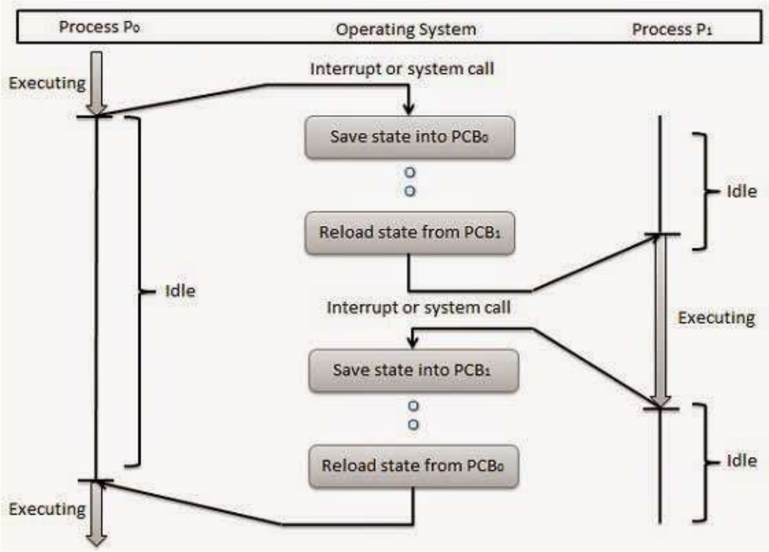
Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Context Switch

- A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.
- When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process.
- Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved. Content switching times are highly dependent on hardware support. Context switch requires (n + m) bxK time units to save the state of the processor with n general registers, assuming b are the store operations are required to save n and m registers of two process control blocks and each store instruction requires K time units.



Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time. When the process is switched, the following information is stored.

- Program Counter
- Scheduling Information
- Base and limit register value
- Currently used register
- Changed State
- I/O State
- Accounting

CONCURRENT AND PROCESSES

Co-operation of Process

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process. There are several reasons for providing an environment that allows process cooperation:

• **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file).It must provide an environment to allow concurrent access to such information.

• **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

• **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads,

• **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an inter-process communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of inter-process communication: shared memory and message passing. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

CRITICAL SECTION PROBLEM

Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

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

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

1. **Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

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

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

Two general approaches are used to handle critical sections in operating systems: preemptive kernels and non-preemptive kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

SEMAPHORE

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities. A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). The wait() operation was originally termed P (from the Dutch proberen, “to test”); signal() was originally called V (from verhogen, “to increment”). The definition of wait() is as follows:

```
wait(S) {
while (S <= 0)
; // busy wait
S--;
}
```

The definition of signal() is as follows:

```
signal(S) {
```

```
S++;
}
```

All modifications to the integer value of the semaphore in the wait () and signal () operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.

Semaphore Usage

Operating systems often distinguish between **counting and binary semaphores**. The value of a **counting semaphore** can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Thus, **binary semaphores** behave similarly to mutex locks. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal () operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Deadlock and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked. To illustrate this, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1: P0 P1

```
wait(S); wait(Q);
wait(Q); wait(S);
. .
. .
. .
signal(S); signal(Q);
signal(Q); signal(S);
```

Suppose that P0 executes wait(S) and then P1 executes wait(Q).When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Since these signal() operations cannot be executed, P0 and P1 are deadlocked. We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

The events with which we are mainly concerned here are resource acquisition and release Another problem related to deadlocks is indefinite blocking or starvation, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

METHODS OF INTER-PROCESS COMMUNICATION (IPC)

Inter-process communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system’s control.

The processes are also responsible for ensuring that they are not writing to the same location simultaneously. Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-Passing Systems

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages. A message-passing facility provides at least two operations:

send(message) receive(message)

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system common kind of tradeoff seen throughout operating-system design. If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network, but rather with its logical implementation. Here are several methods for logically implementing a link and the send()/receive() operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication. Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

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

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send () and receive () primitives are defined as follows:

• send (P, message)—Send a message to process P.

• receive (id, message)—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection.

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The send () and receive () primitives are defined as follows:

• send (A, message)—Send a message to mailbox A.

• receive (A, message)—Receive a message from mailbox A.

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

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Now suppose that processes P1, P2, and P3 all share mailbox A. Process P1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1? The answer depends on which of the following methods we choose:

• Allow a link to be associated with two processes at most.

• Allow at most one process at a time to execute a receive () operation.

• Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message).

The system may define an algorithm for selecting which process will receive the message (for example, round robin, where processes take turns receiving messages). The system may identify the receiver to the sender. A mailbox may be owned either by a process or by the operating system.

If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears.

Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists. In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

• Create a new mailbox.

• Send and receive messages through the mailbox.

• Delete a mailbox.

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

Synchronization

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either blocking or non blocking— also known as synchronous and asynchronous. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

- Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox.
- Non-blocking send. The sending process sends the message and resumes operation.
- Blocking receive. The receiver blocks until a message is available.
- Non-blocking receive. The receiver retrieves either a valid message or a null.

Different combinations of send () and receive () are possible. When both send () and receive () are blocking, we have a rendezvous between the sender and the receiver. The solution to the producer–consumer problem becomes trivial when we use blocking send () and receive () statements. The producer merely invokes the blocking send () call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive (), it blocks until a message is available.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- Zero capacity. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- Bounded capacity. The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

• Unbounded capacity. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

DEADLOCK

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

1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
3. Release. The process releases the resource.

- For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure 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.
- A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, semaphores, mutex locks, and files).

Deadlock Prevention

Mutual Exclusion

➢ The mutual exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

- A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable. For example, a mutex lock cannot be simultaneously shared by several processes.

Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that we can use requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls. An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.
- Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. In the example given, for instance, we can release the DVD drive and disk file, and then request the disk file and printer, only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the beginning for both protocols.
- Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol.

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting. Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

Circular Wait

- The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- To illustrate, we let R = {R1, R2, ..., Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12

- We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type —say, Ri . After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri). For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.
- Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that F(Ri) ≥ F(Rj). Note also that if several instances of the same resource type are needed, a single request for all of them must be issued. If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be {P0, P1, ..., Pn}, where Pi is waiting for a resource Ri , which is held by process Pi+1. (Modulo arithmetic is used on the indexes, so that Pn is waiting for a resource Rn held by P0.) Then, since process Pi+1 is holding resource Ri while requesting resource Ri+1, we must have F(Ri) < F(Ri+1) for all i. But this condition means that F(R0) < F(R1) < ... < F(Rn) < F(R0). By transitivity, F(R0) < F(R0), which is impossible. Therefore, there can be no circular wait.

Deadlock Avoidance:

- For avoiding deadlocks, it is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive.
- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

- The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

Safe State:

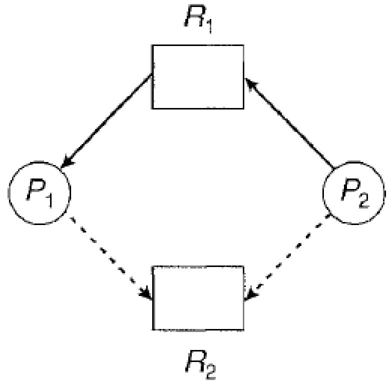
- A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.
- A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When Pi terminates, Pi+l can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe. A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state.
-

Resource-Allocation-Graph Algorithm

- If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph defined for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge.
- A claim edge Pi ~ Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process Pi requests resource R1, the claim edge P; -+ R1 is converted to a request edge. Similarly, when a resource R1 is released by P;, the assignment edge Rj -+ P; is reconverted to a claim edge P; -+ Rj.
- We note that the resources must be claimed a priori in the system. That is, before process P; starts executing, all its claim edges must already appear in the resource-allocation

graph. We can relax this condition by allowing a claim edge P; -+ R1 to be added to the graph only if all the edges associated with process P; are claim edges. Now suppose that process P; requests resource Rj. The request can be granted only if converting the request edge P; -+ Rj to an assignment edge R1 -+ P; does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm.

- An algorithm for detecting a cycle in this graph requires an order of n2 operations, where n is the number of processes in the system. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process P; will have to wait for its requests to be satisfied. To illustrate this algorithm, we consider the following resource-allocation graph.



- Suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this action will create a cycle in the graph .A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

Banker’s algorithm:

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The Banker's algorithm is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm. When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This nun1.ber may not exceed the total number of resources in the system.

- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources. Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Ri are available.

Max: An n x m matrix defines the maximum demand of each process. If Max[i] [j] equals k, then process P; may request at most k instances of resource type Ri.

Allocation: An 11 x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals lc, then process P; is currently allocated lc instances of resource type Rj.

Need: An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process P; may need k more instances of resource type Ri to complete its task. Note that Need[i][j] equals Max[i][j] - Allocation [i][j].

These data structures vary over time in both size and value. To simplify the presentation of the banker's algorithm, we next establish some notation. Let X andY be vectors of length 11. We say that X::= Y if and only if X[i] ::= Y[i] for all i = 1, 2, ... , n. For example, if X = (1,7,3,2) and Y = (0,3,2,1), then Y ::=X. In addition, Y < X if Y ::=X and Y# X. We can treat each row in the matrices Allocation and Need as vectors and refer to them as Allocation; and Need;. The vector Allocation; specifies the resources currently allocated to process P;; the vector Need; specifies the additional resources that process P; may still request to complete its task.

Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work= Available and Finish[i] =false for i = 0, 1, ... , n - 1.

2. Find an index i such that both
    a. Finish[i] ==false
    b. Need$_i$ Work. If no such i exists, go to step 4.
3. Work = Work + Allocation; Finish[i] = true. Go to step 2.
4. If Finish[i] ==true for all i, then the system is in a safe state.

Resource-Request Algorithm

Let Request; be the request vector for process P;. If Request; [j] == k, then process P; wants k instances of resource type Rj. When a request for resources is made by process P;, the following actions are taken:

1. If Request; <= Need; go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request< = Available, go to step 3. Otherwise, P; must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P; by modifyil1.g the state as follows:

   Available= Available- Request$_i$

Deadlock

Allocation; =Allocation; +Request$_i$

Need; =Need$_i$- Request$_i$

If the resulting resource-allocation state is safe, the transaction is completed, and process P; is allocated its resources. However, if the new state is unsafe, then P; must wait for Request$_i$, and the old resource-allocation state is restored.

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

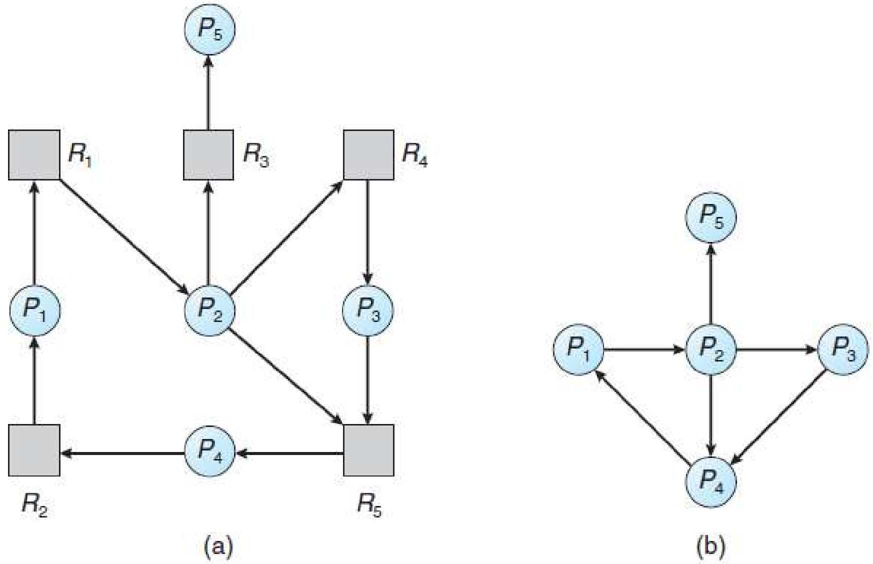
• An algorithm that examines the state of the system to determine whether a deadlock has occurred

• An algorithm to recover from the deadlock

Single Instance of Each Resource Type

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

More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.



An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq . In Figure, we present a resource-allocation graph and the corresponding wait-for graph. As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.

To detect deadlocks, the system needs to maintain the wait for graph and periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm

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

• Allocation. An n × m matrix defines the number of resources of each type currently allocated to each process.

• Request. An n × m matrix indicates the current request of each process. If Request[i][j] equals k, then process Pi is requesting k more instances of resource type Rj . To simplify notation, we again treat the rows in the matrices Allocation and Request as vectors; we refer to them as Allocationi and Requesti . The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available. For i = 0, 1, ..., n–1, if Allocationi _= 0, then Finish[i] = false. Otherwise, Finish[i] = true.

2. Find an index i such 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, 0≤i<n, then the system is in a deadlocked state. Moreover, if Finish[i] == false, then process Pi is deadlocked. This algorithm requires an order of m × n2 operations to detect whether the system is in a deadlocked state. You may wonder why we reclaim the resources of process Pi (in step 3) as soon as we determine that Requesti ≤ Work (in step 2b). We know that Pi is currently not involved in a deadlock (since Requesti ≤ Work). Thus, we take an optimistic attitude and assume that Pi will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T0, we have the following resource-allocation state:

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <P0, P2, P3, P1, P4> results in Finish[i] == true for all i. Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

	Request
	A B C
P0	0 0 0
P1	2 0 2
P2	0 0 1
P3	1 0 0
P4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock.

Process Termination

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

• Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

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

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2. Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as

necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

POSSIBLE QUESTIONS

UNIT – II

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. Define internal commands in Unix.
2. What is External commands in Linux.
3. What do you mean by partition in OS?
4. Define Fork command.
5. Define Wait commands

PART – C (6 MARKS)

1. Discuss in detail the System Processes.
2. Explain the external and internal commands.
3. Explain the creation of partitions in OS.
4. Explain the processes and its creation phases.
5. Discuss the commands of fork, exec and wait.

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - II : (Objective Type Multiple choice Questions each Question carries one Mark)

UNIX/LINUX PROGRAMMING

PART - A (Online Examination)

Questions	Opt1	opt2	opt3	opt4	KEY
Semaphores function is to	synchronize critical resources to prevent deadlock	synchronize processes for better CPU utilization	used for memory management	may cause a high I/O rate	synchronize critical resources to prevent
Four necessary conditions for deadlock are non pre-emption, circular wait, hold and wait and	mutual exclusion	race condition	buffer overflow	multiprocessing	mutual exclusion
A series of statements explaining how the data is to be processed is called	instruction	compiler	program	interpretor	program
Banker's algorithm deals with	deadlock prevention	deadlock avoidance	deadlock recovery	mutual exclusion	deadlock avoidance
Which is non pre-emptive	Round robin	FIFO	MQS	MQSF	FIFO
A hardware device which is capable of executing a sequence of instructions, is known as	CPU	ALU	CU	Processor	Processor
Distributed systems should	high security	have better resource sharing	better system utilization	low system overhead	have better resource sharing
Which of the following is always there in a computer	Batch system	Operating system	Time sharing system	Controlling system	Operating system

Which of following is not an advantage of multiprogramming	increased throughput	shorter response time	ability to assign priorities of jobs	decreased system overload	decreased system overload
Banker's algorithm for resource allocation deals with	deadlock prevention	deadlock aviodance	deadlock recovery	circular wait	deadlock aviodance
_______ is the basis of multiprogrammed operating system.	RR scheduling	Self Scheduling	CPU sceduling	throughput	CPU sceduling
A ______ is executed until it must wait, typically for the completion of some i/o request	reverse	deadlock avoidance	deadlock	process	process
_____ is a fundamental operating system function.	RR	CPU	Scheduling	nonpreemptive	Scheduling
Process execution begins with a_____	CPU burst	RR scheduling	SJF scheduling	SRT scheduling	CPU burst
The operating system must select one of the processes in the ready queue to be executed by the __________	nonpreemptive	short term scheduler	long term scheduler	low level	short term scheduler
When scheduling takes place only under circumstances 1 and 4 called _______	variable class	real time class	priority class	nonpreemptive	nonpreemptive
Another component involved in the CPU scheduling function is the ______	central edge	dispatcher	claim edge	graph edge	dispatcher
One measure of work is the number of processes completed per time unit called _____	throughput	variable class	real time class	priority class	throughput
Which of the following is the simplest scheduling discipline?	FCFS scheduling	RR scheduling	SJF scheduling	SRT scheduling	FCFS scheduling
In which scheduling, processes are dispatched according to their arrival time on the ready queue?	FCFS scheduling	RR scheduling	SJF scheduling	SRT scheduling	FCFS scheduling
In which scheduling, processes are dispatched FIFO but are given a limited amount of CPU time?	FIFO scheduling	RR scheduling	SJF scheduling	SRT scheduling	RR scheduling
Which scheduling is effective in time sharing environments	FIFO scheduling	RR scheduling	SJF scheduling	SRT scheduling	RR scheduling

Variable size blocks are called	Pages	Segments	Tables	None	Segments
Which scheduling is effective in time sharing environments	FIFO scheduling	RR scheduling	SJF scheduling	SRT scheduling	RR scheduling
Which of the following is non-preemptive scheduling?	RR scheduling	SJF scheduling	SRT scheduling	None	SJF scheduling
The interval from the time of submission of a process to the time of completion is the ____	Queues	Processor Sharing	Sharing resources	turaround time	turaround time
The simplest CPU sceduling algorithm is the	FCS	SJS	FCFS	DFG	FCFS
The SJF algorithm is a special case of the general _______ algorithm	FCS	SJS	Roundrobin	FCSC	Roundrobin
_______ scheduling algorithm is designed especially for time sharing systems.	CFS	FSCS	priority	Round Robin	Round Robin
The seek optimization strategy in which there is no reordering of the queue is called __________.	FCFS	SSTF	SCAN	C-SCAN	FCFS
A major problem with priority scheduling algorithms is _____	tail	Starvation	time first	time quantum	Starvation
If the time quantum is very small the RR aproach is called ________	Queues	Processor Sharing	Sharing resources	Context switching	Processor Sharing
The seek optimization strategy in which the disk arm is positioned next at the request (inward or outward) that minimizes arm movement is called ________.	FCFS	SSTF	SCAN	C-SCAN	SSTF
If several identical processors are available then _____ can occur.	heterogeneous	homogeneous	load sharing	UMA	load sharing
The high priority process would be waiting for a lower priority one to finish is called _____	resources inversion	Priority inversion	priority	Priority inheritance	Priority inversion
_____ systems are required to complete a critical task within a guaranteed amount of time.	hard real time	Priority inversion	load sharing	Priority inheritance	hard real time
The scheduler than either admits a process guarenteeing that the process will complete on time known as _____	Priority inversion	resources reservation	load sharing	Sharing resources	resources reservation

_______ uses the the given algorithm and the system workload to produce a formula.	deterministic modelling	scheduling process	Analaytic evaluation	Queuing model	Analaytic evaluation
If no thread is found the dispatcher will execute a special thread called______	variable class	real time class	priority class	idle thread	idle thread
Deadlocks can be described more precisely in terms of a directed graph called	resource graph	system graph	system resources allocation graph	request graph	system resources allocation graph
______ is th set of methods for ensuring that at atleast one of the necessary condition.	Deadlock prevention	deadlock avoidance	handling deadlock	resource deadlock	Deadlock prevention
_____ is possible to construct an algorithm that ensures that the system will never enter the deadlock state.	Deadlock prevention	deadlock avoidance	handling deadlock	resource deadlock	deadlock avoidance
A system is in a safe state only if there exists a ______	Safe state	unsafe state	normal	deadlock	Safe state
A critical section is a program segment _____________.	which should run in a certain specified amount	which avoids deadlocks	where shared resources are accessed	which must be enclosed by a pair of semaphore operations, P and V	where shared resources are accessed
The deadlock avoidance algorithm are described in next system but is less efficient than the resource allocation graph called ______	Deadlock prevention	deadlock avoidance	bankers algorithm	bankers allocation	bankers algorithm
CPU Scheduling is the basis of _________ operating system.	single programmed	multi programmed	multi system	multi disks	multi programmed
Scheduling is a fundamental ___________ function.	computer	operating system	system resource	disk	operating system
Another component involved in the CPU _______ function is the dispatcher	processing	mathematical	arithmetic	scheduling	scheduling
A major problem for priority __________ is starvation.	sort algorithms	scheduling algoriuthms	search algorithms	manage algorithms	scheduling algoriuthms

The seek _______ strategy in which there is no reordering of the queue is called SSTF	processing	scheduling	optimization	implementation	optimization
The high ___________ would be waiting for a lower priority one to finish is called priority inversion	performance	priority	patent	graph edge	priority
A __________ is a program segment where shared resources are accessed.	critical section	sub section	cross section	class section	critical section
If no thread is found, the ________ will execute a special thread called idle thread.	degrader	scheduler	dispatcher	redeemer	dispatcher
_______ execution begins with a CPU Burst.	Process	Performance	Purge	Put	Process
SJF Scheduling is an example for _________ scheduling.	non- preemptive	preemptive	emptive	prescheduling	non-preemptive
_______ is the simplest CPU sceduling algorithm.	FCFS	LCFS	FCLS	LCFS	FCFS
Segments are called ________ blocks.	equal size	variable class	variable size	big size	variable size
________ can be described more precisely in terms of a directed graph.	semaphore	deadlocks	dumplocks	starvation	deadlocks
The interval from the time of submission of a process to the time of completion is the ____	Queues	Processor Sharing	Sharing resources	turaround time	turaround time

UNIT – III

SYLLABUS

User Management and the File System Types of Users, Creating users, Granting rights User management commands, File quota and various file systems available, File System Management and Layout, File permissions, Login process, Managing Disk Quotas, Links (hard links, symbolic links)

- The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services.
- Consider, for example, that an operating system for a multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

Address Binding

- Memory management is the functionality of an operating system which handles or manages primary memory. Memory management keeps track of each and every memory location either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.
- Memory management provides protection by using two registers, a base register and a limit register. The base register holds the smallest legal physical memory address and the limit register specifies the size of the range. For example, if the base register holds 300000 and the limit register is 1209000, then the program can legally access all addresses from 300000 through 411999.

Instructions and data to memory addresses can be done in following ways

- Compile time -- When it is known at compile time where the process will reside, compile time binding is used to generate the absolute code.
- Load time -- When it is not known at compile time where the process will reside in memory, then the compiler generates re-locatable code.
- Execution time -- If the process can be moved during its execution from one memory segment to another, then binding must be delayed to be done at run time

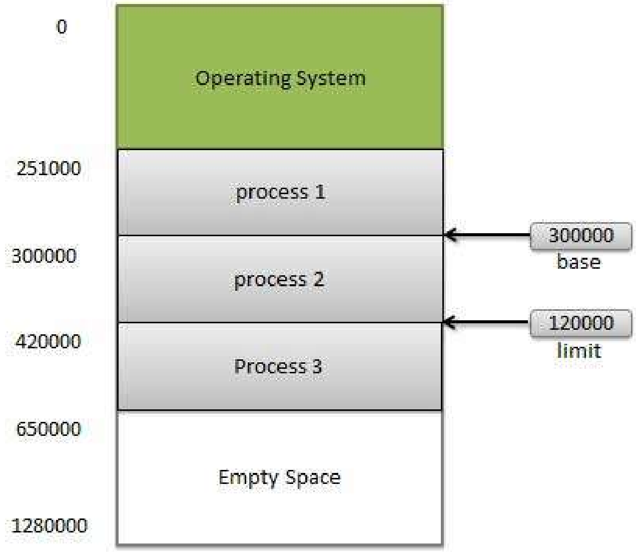
Dynamic Loading

- In dynamic loading, a routine of a program is not loaded until it is called by the program. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed. Other routines methods or modules are loaded on request. Dynamic loading makes better memory space utilization and unused routines are never loaded.
- The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

Dynamic Linking

- Linking is the process of collecting and combining various modules of code and data into a executable file that can be loaded into memory and executed. Operating system can link system level libraries to a program. When it combines the libraries at load time, the linking is called static linking and when this linking is done at the time of execution, it is called as dynamic linking.

- In static linking, libraries linked at compile time, so program code size becomes bigger whereas in dynamic linking libraries linked at execution time so program code size remains smaller.
- Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.



PHYSICAL AND VIRTUAL ADDRESS SPACE

Logical (Virtual) versus Physical Address Space

An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address. Logical address is also known a Virtual address. Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

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

MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

MEMORY ALLOCATION STRATEGIES

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.
- We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Protection

- Before discussing memory allocation further, we must discuss the issue of memory protection. If we have a system with a relocation register, together with a limit, we accomplish our goal. The relocation register contains the value of the smallest physical

address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).

- Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple **partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management.
- In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes.
- As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.

- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- In general, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes. This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

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

• **Best fit.** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

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

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation.** As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory

space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

- Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.
- One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. Two complementary techniques achieve this solution: segmentation and paging

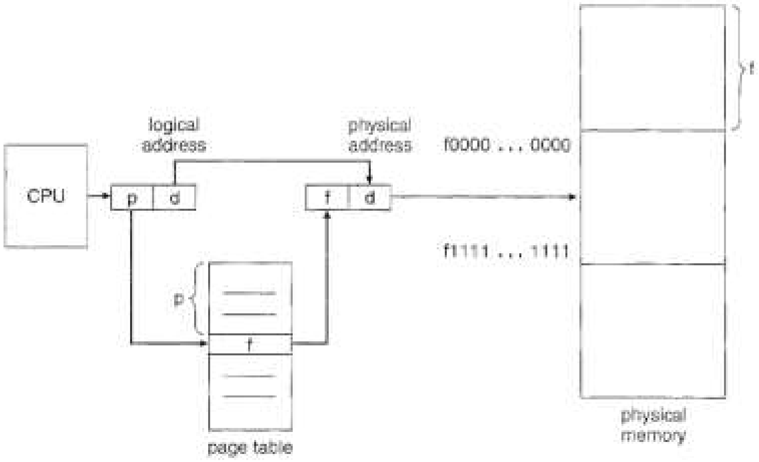
PAGING

- It is a memory-management scheme that permits the physical address space a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be framed on the backing store.
- The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems.

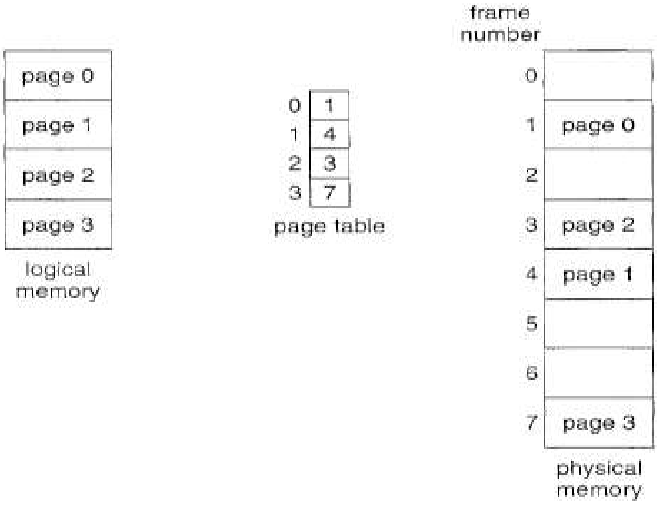
  Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

Basic Method

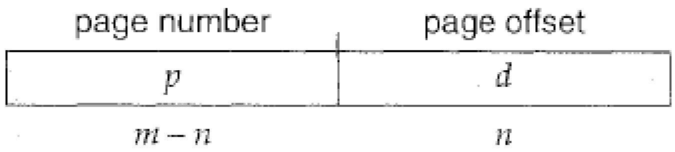
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are of the san1.e size as the memory frames. The hardware support for paging is illustrated in the following figure



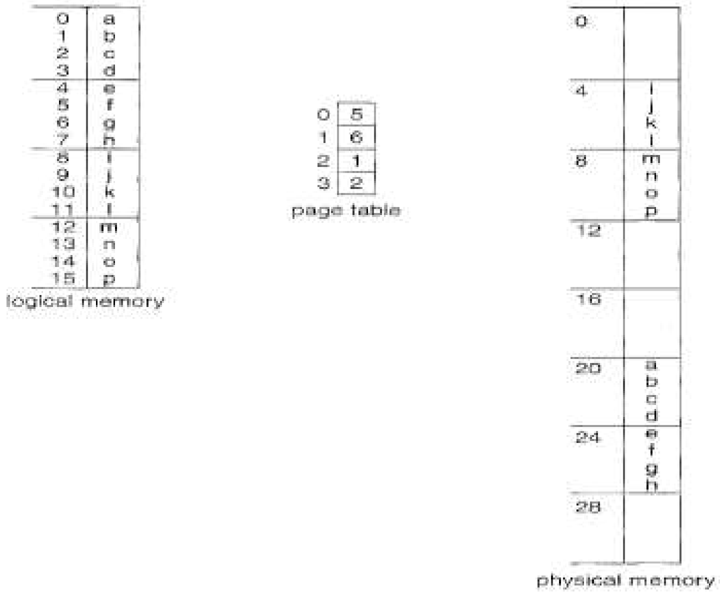
- Every address generated the CPU is divided into two parts: a {p) and a . The page number is used as an index into a page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in the following diagram



- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.



- If the size of the logical address space is 2m, and a page size is 271 addressing units (bytes or wordst then the high-order m- n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows: where p is an index into the page table and d is the displacement within the page. As a concrete (although minuscule) example, consider the memory in the following diagram



- Here, in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 x 4) + O]. Logical address 13 maps to physical address 9. You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address.

Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory. When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units.

- If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 - 1,086 = 962 bytes. In the worst case, a process would need 11 pages plus 1 byte. It would be allocated 11 + 1 frames, resulting in internal fragmentation of almost an entire frame. If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.
- Today, pages typically are between 4 KB and 8 KB in size and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages. Researchers are now developing support for variable on-the-fly page size. Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 232 physical page frames. If frame size is 4 KB, then a system with 4-byte entries can address 244 bytes (or 16 TB) of physical memory. When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires 11 pages, at least 11 frames must be available in memory.
- If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on. An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views

memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.

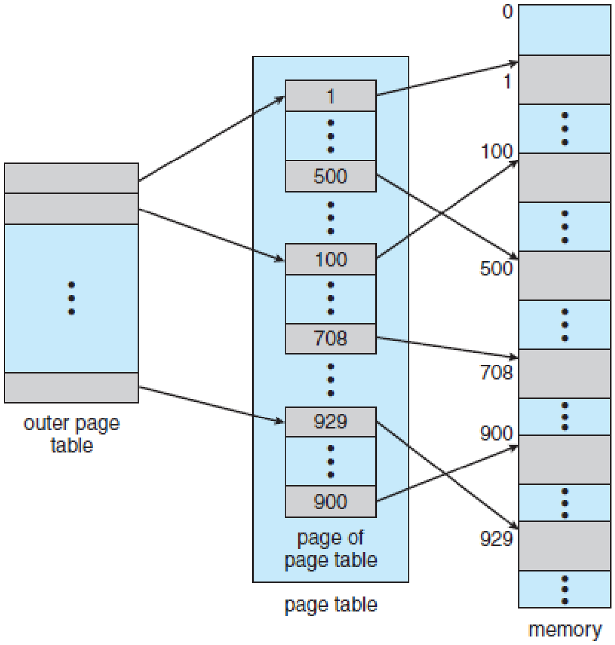
- The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own.
- It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns. Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory-which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame the frame-table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.
- In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses.If a user makes a system call (to do I/0, for example) and provides an address as a parameter (a buffe1~ for instance), that address must be mapped to produce the correct physical address.
- The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

STRUCTURE OF PAGE TABLE

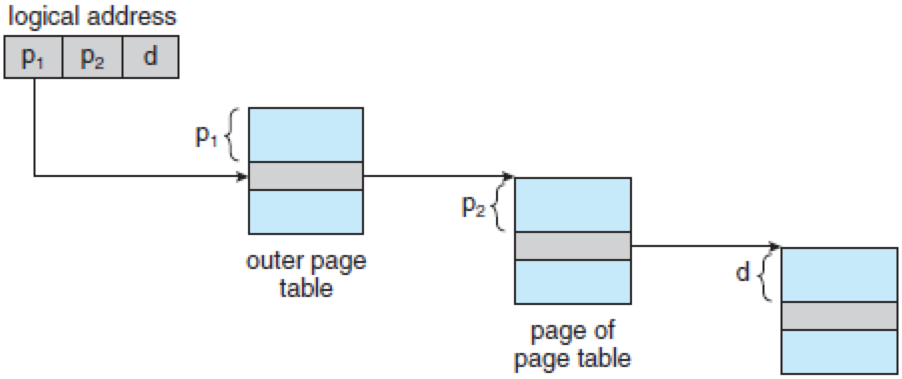
In this section, we explore some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

Hierarchical Paging

- Most modern computer systems support a large logical address space (232 to 264). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (212), then a page table may consist of up to 1 million entries (232/212). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces.



- We can accomplish this division in several ways. One way is to use a two-level paging algorithm, in which the page table itself is also paged. For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number

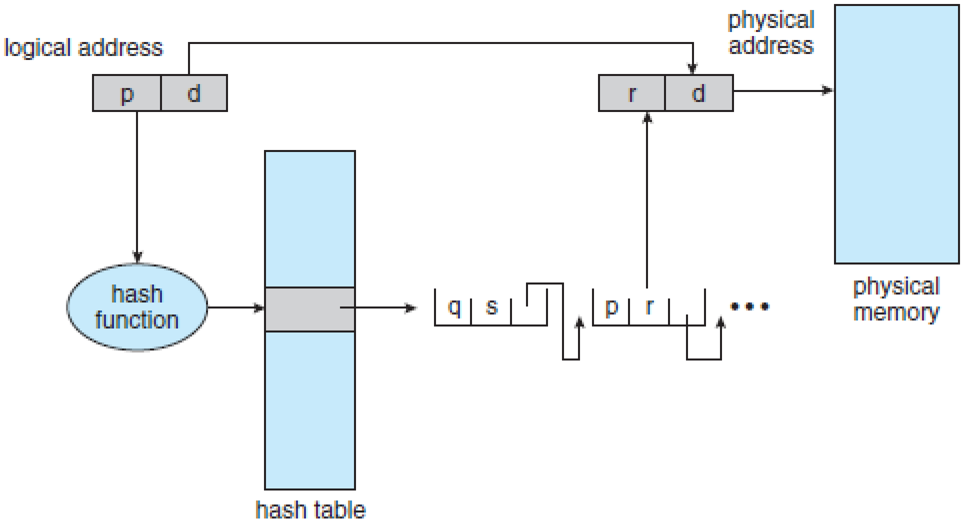


consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

Where p1 is an index into the outer page table and p2 is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure. Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

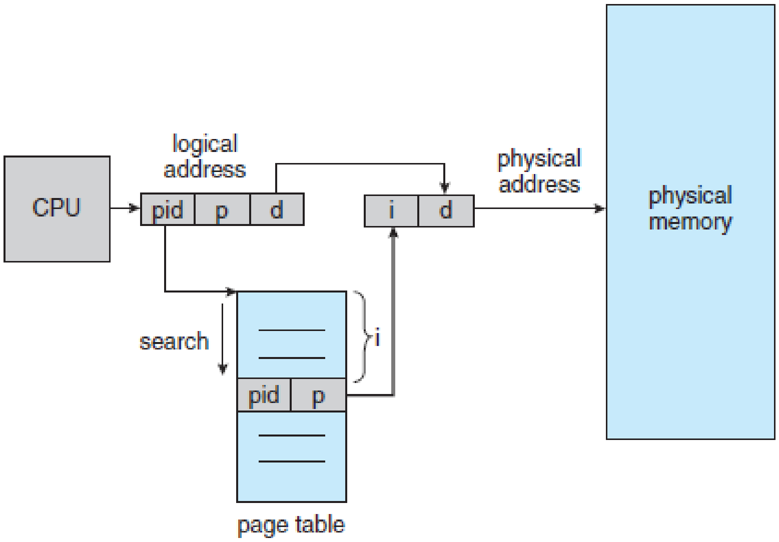
Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list. The algorithm works as follows:
- The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



Inverted Page Tables

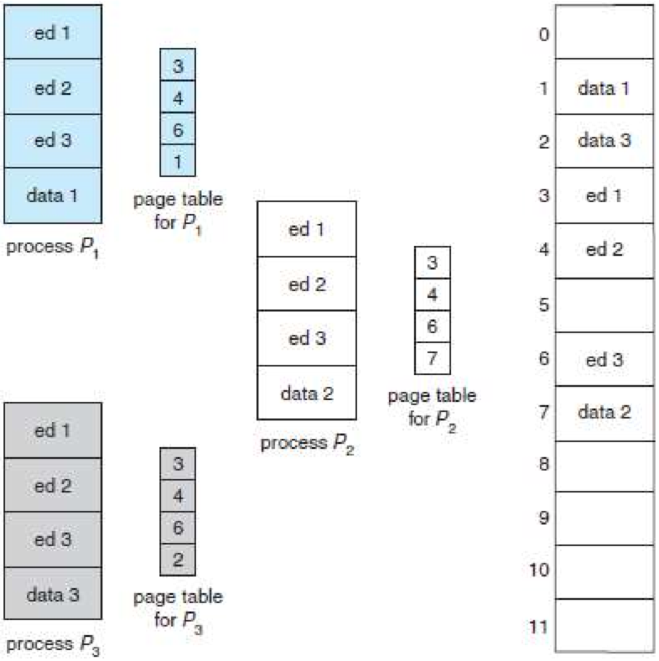
- Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses.
- The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.



- To solve this problem, we can use an inverted page table. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

Shared Pages

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code (or pure code), it can be shared, as shown in Figure. Here, we see three processes sharing a three-page editor—each page 50 KB in size (the large page size is used to simplify the figure). Each process has its own data page. Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.



- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings. Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.
- The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads.

SEGMENTATION

- An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory from the actual physical memory. As we have already seen, the user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. This mapping allows differentiation between logical memory and physical memory.

Basic Methods

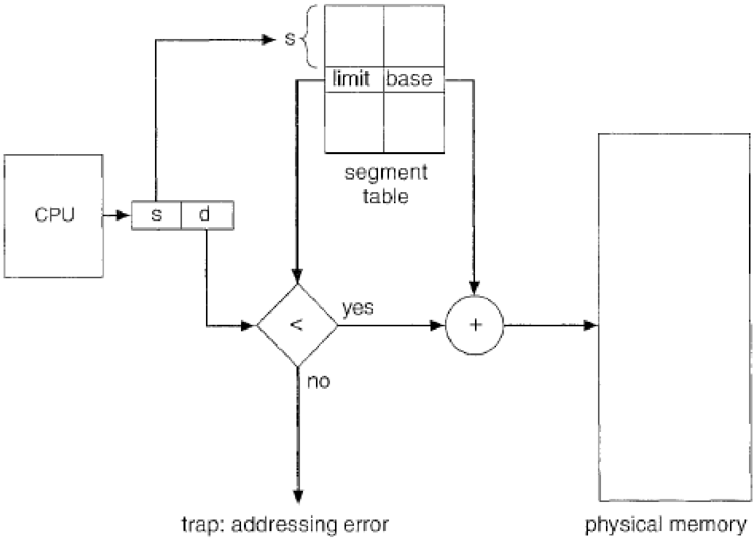
- It is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.) For simplicity of implementation, segments are numbered and are referred to by a segn"lent number, rather than by a segment name. Thus, a logical address consists of a two tuple:

  <segment-number, offset>.

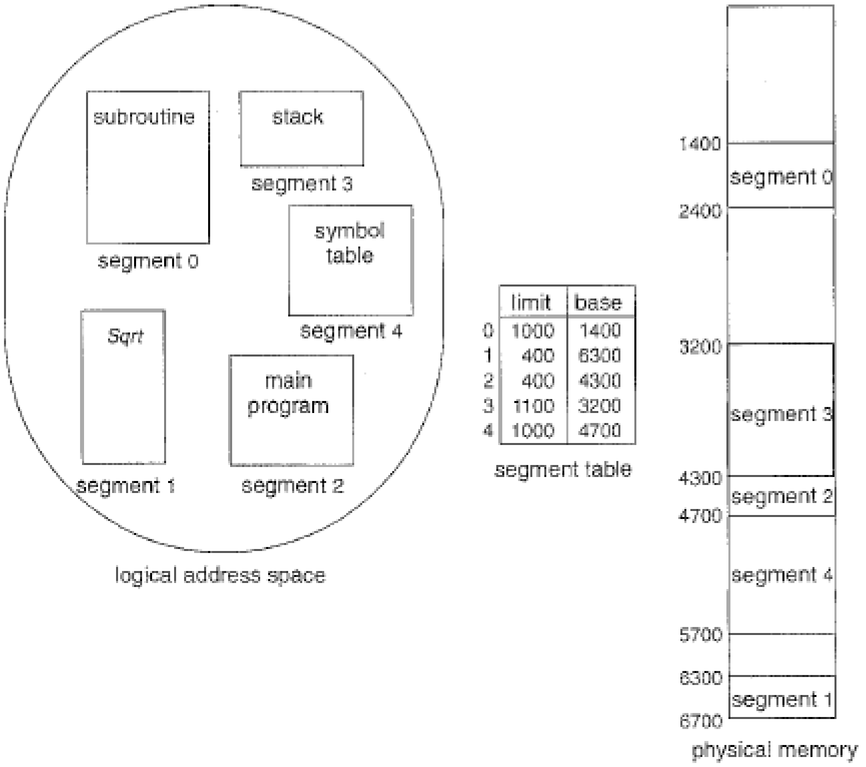
- Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A C compiler might create separate segments for the following:
  1. The code
  2. Global variables
  3. The heap, from which memory is allocated
  4. The stacks used by each thread
  5. The standard C library
- Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Hardware

- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by each entry in the segment table has a segment base and a segment limit. The segment base contains the start physical address where the segment resides in memory, and the segment limit specifies the length of the segment. The use of a segment table is illustrated in Figure



- A logical address consists of two parts: a segment number, s, and an offset into that segment, d. the segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs. As an example, consider the situation shown in Figure



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

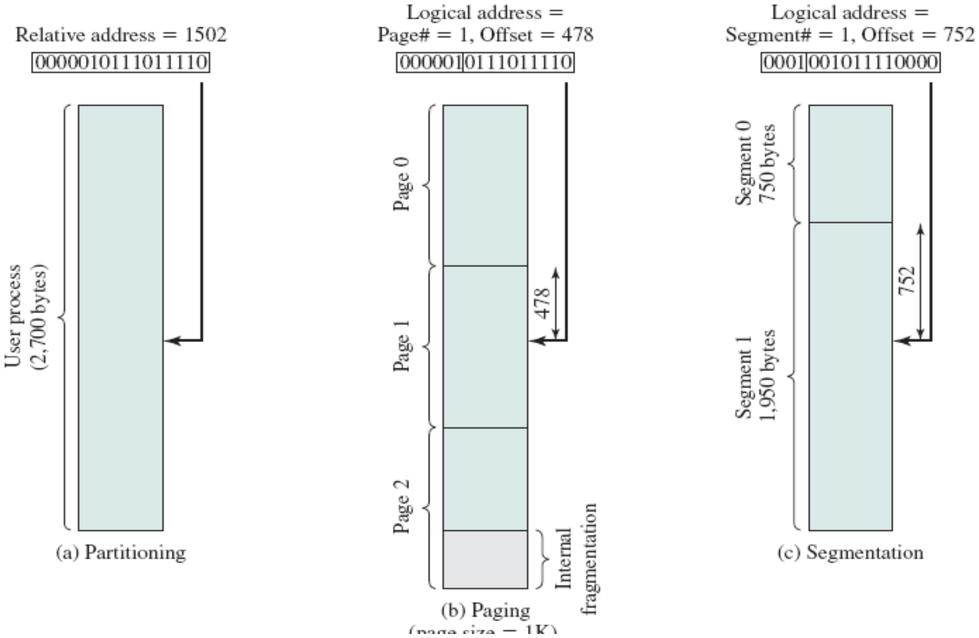
- We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

Segmentation and Paging

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

segment number and an offset. Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning.

- In the absence of an overlay scheme or the use of virtual memory, it would be required that all of a program's segments be loaded into memory for execution. The difference, compared to dynamic partitioning, is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation.
- However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less. Whereas paging is invisible to the programmer, segmentation is usually visible and is provided as a convenience for organizing programs and data. STypically, the programmer or compiler will assign programs and data to different segments. For purposes of modular programming, the program or data may be further broken down into multiple segments.
- The principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation. Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses.
- Each segment table entry would have to give the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment, to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware. Consider an address of n_m bits, where the leftmost n bits are the segment number and the rightmost m bits are the offset. In our example (Figure C), n _ 4 and m _ 12. Thus the maximum segment size is 2 12 _ 4096.



(a) Partitioning

(b) Paging
(page size = 1K)

(c) Segmentation

The following steps are needed for address translation:

- Extract the segment number as the leftmost n bits of the logical address.
- Use the segment number as an index into the process segment table to find the starting physical address of the segment.
- Compare the offset, expressed in the rightmost m bits, to the length of the segment.

If the offset is greater than or equal to the length, the address is invalid. The desired physical address is the sum of the starting physical address of the segment plus the offset.

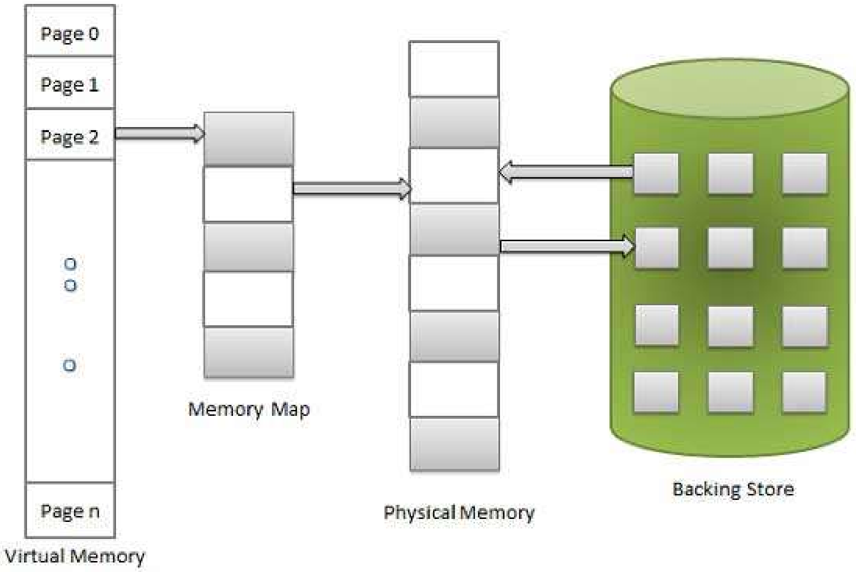
In our example, we have the logical address 0001001011110000, which is segment number 1, offset 752. Suppose that this segment is residing in main memory starting at physical address 0010000000100000. Then the physical address is 0010000000100000 + 001011110000 _ 0010001100010000.

To summarize, with simple segmentation, a process is divided into a number of segments that need not be of equal size. When a process is brought in, all of its segments are loaded into available regions of memory, and a segment table is set up.

VIRTUAL MEMORY

Virtual Memory and its Organization

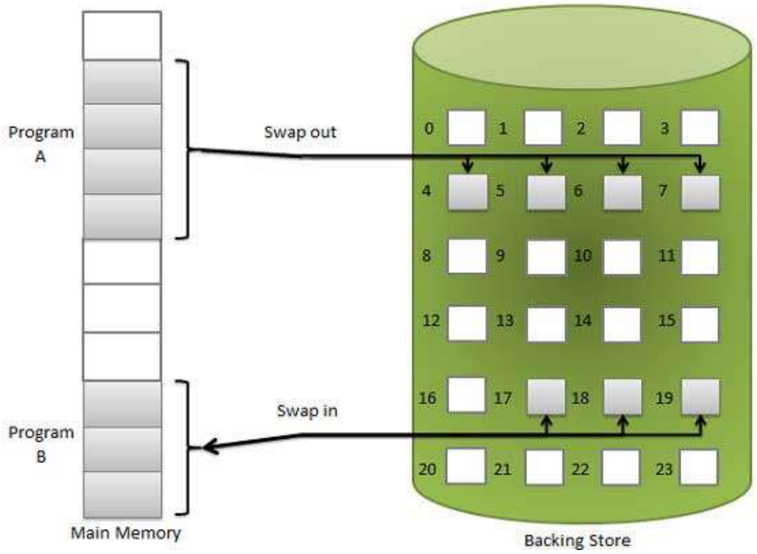
- Virtual memory is a technique that allows the execution of processes which are not completely available in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Following are the situations, when entire program is not required to be loaded fully in main memory.
- User written error handling routines are used only when an error occured in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.



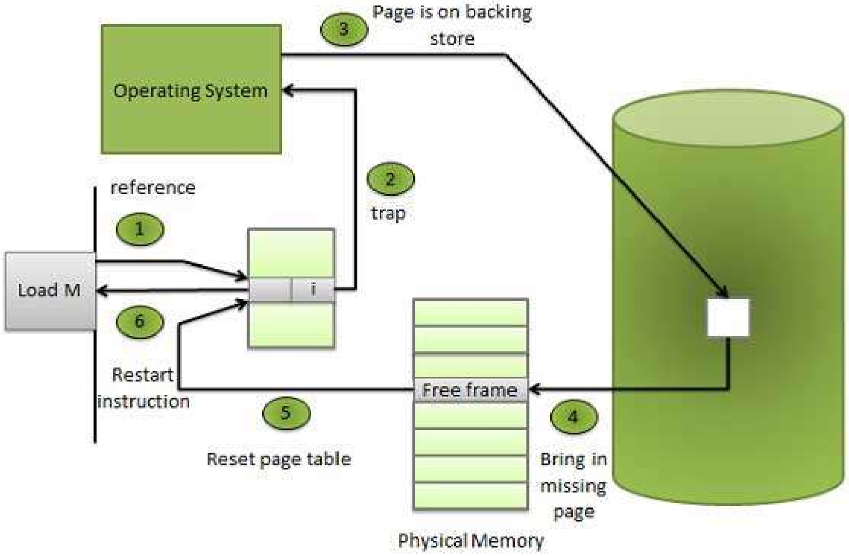
- Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

- A demand paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper called pager.
- When a process is to be swapped in, the pager, guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.
- Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme, where valid and invalid pages can be checked by checking the bit. Marking a page will have no effect if the process never attempts to access the page. While the process executes and accesses pages that are memory resident, execution proceeds normally.



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



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

- **Advantages**

Following are the advantages of Demand Paging

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

- **Disadvantages**

Following are the disadvantages of Demand Paging

- Number of tables and amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.
- Due to the lack of an explicit constraint on a job address space size.

POSSIBLE QUESTIONS

UNIT – III

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What do you mean by granting right?
2. What is user management commands?
3. What is File Quota?
4. Define File Layout.
5. What do you mean by managing disk quotas?
6. Define symbolic link.

PART – C (6 MARKS)

1. Explain the types of users.
2. Explain the user management commands.
3. Explain the file quota and various file systems available.
4. Discuss the file system management and layout.
5. Explain the file permissions.
6. Discuss the login process.

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - III : (Objective Type Multiple choice Questions each Question carries one Mark)

UNIX/LINUX PROGRAMMING

PART - A (Online Examination)

Questions	Opt1	opt2	opt3	opt4	KEY
Memory is array of ___________	bytes	circuits	ics	ram	bytes
CPU fetches instructions from ______	memory	pendrive	dvd	cmos	memory
Program must be in ________________	dvd	pendrive	memory	cmos	memory
Collection of process in disk forms_____	input queue	output queue	stack	circle	input queue
Address space of computer starts at _________	3333	4444	0000	2222	0000
If process location is found during compile time then _________ code is generated	relative	absolute	approximate	more or less	absolute
Address generated by CPU is ______________ address	logical	physical	direct	indirect	logical
Logical address can be also called as ______________ address	physical	virtual	direct	indirect	virtual
Run time mapping is done using _____________	MMU	CPU	CU	IU	MMU
In address binding base register is also called as ______________	relocation register	memory register	hard disk	pendrive	relocation regis

Better memory space is utilized using ________________	dynamic loading	dynamic linking	registers	array of words	dynamic loadin
_________________ routine is never loaded in dynamic loading	unused	used	regular	recursive	unused
Some operating systems support only __________ linking	static	dynamic	temporary	interruptive	static
______________ is a code that locates library routine	stub	dll	recursive routine	exe file	stub
____________ can be used to manage large memory requirement for a process	overlays	swapping	roll in and out	libraries	overlays
______________ error is raised in memory	addressing	swapping	dynamic	index	addressing
Set of ___________ are scattered throughout the memory	holes	gaps	free space	words	holes
_____________ can be internal and external	fragmentation	merging	grouping	fixing	fragmentation
____________ is used to divide a process into fixed size chunks	paging	segmentation	sp	swapping	paging
In paging physical memory is divided into _______________	frames	pages	segments	bytes	frames
In paging virtual memory is divided into _______________	frames	pages	segments	bytes	pages
_____________ is first of virtual address in paging	page number	segment number	frame number	offset	page number
_____________ is second part of virtual address in paging	page number	segment number	frame number	offset	offset
Page mapping entries are found in __________	page table	segment table	hash table	pointing table	page table
Page size is defined by ______________	hardware	software	os	kernel	hardware
_____________ is first in mapping of virtual to physical address in paging	direct	associate	direct & associative	pointing	direct
_____________ is second in mapping of virtual to physical address in paging	direct	associate	direct & associative	pointing	associate
_____________ is third in mapping of virtual to physical address in paging	direct	associate	direct & associative	pointing	direct & associa

_____________ is used to divide a process into variable size chunks	paging	segmentation	sp	swapping	segmentation
In segmentation virtual memory is divided into ________________	frames	pages	segments	bytes	segments
____________ view is supported in segmentation	user	system	cpu	manager	user
_____________ is format for segmentation virtual address	(s,d)	(p,d)	(v,d)	(k,d)	(s,d)
_____________ is the first element in segment table	limit	base	offset	page number	limit
_____________ is the second element in segment table	limit	base	offset	page number	base
Addressing in segmentation is similar as ____________ addressing in paging	direct	associate	direct & associative	pointing	direct
How many elements are there in segmentation address _________	1	2	3	4	3
_____________ is organization in physical memory in segmentation	frames	pages	segments	bytes	frames
________________ memory is used to manage incompleteness of a process execution	virtual	physical	rom	eprom	virtual
virtual memory abstracts _________ memory	virtual	eerom	main	eprom	main
_______________ reasons are there for existence for virtual memory	1	2	3	4	3
_______________ benefits are there from virtual memory	1	2	3	4	3
Virtual memory is commonly implemented by ____________ paging	demand	bargain	quarrel	order	demand
________________ fault occurs when desired page is not in memory	page	segment	pages	segments	page
_______________ table is used in demand paging	page	segment	pages	segments	page
_______________ methods are there for process creation	1	2	3	4	2

_______________ method implements partial sharing in process creation	copy on write	memory mapping	paging	segmentation	copy on write
_______________ is done for page fault	replacement	swapping	logging	locking	replacement
_______________ is unrealizable page replacement algorithm	optimal	FIFO	LRU	NRU	optimal
_______________ is first page replacement algorithm	optimal	FIFO	LRU	NRU	FIFO
_______________ is second page replacement algorithm	optimal	FIFO	LRU	NRU	optimal
_______________ is third page replacement algorithm	optimal	FIFO	LRU	NRU	NRU
_______________ is associated with each page in optimal algorithm	label	index	number	identity	label
_______________ labelled page replaced in optimal algorithm	highest	lowest	moderate	below average	highest
_______________ end page is removed in fifo algorithm	rear	head	top	bottom	head
Modified version of fifo algorithm gives ____________ chance to a page	1	2	3	4	2
______________ is called as high paging activity	thrashing	smashing	mocking	breaking	thrashing
______________ occurs frequently during thrashing	page fault	segment fault	memory fault	address fault	page fault
_______________ strategy is used to solve thrashing a little	working set	pff	lpr algorithm	gpl algorithm	working set
_______________ algorithm is used to solve thrashing a little	working set	pff	lpr algorithm	gpl algorithm	lpr algorithm
_______________ is a basic solution for thrashing	working set	pff	lpr algorithm	gpl algorithm	pff

ter

g

ative

UNIT – III

SYLLABUS

User Management and the File System Types of Users, Creating users, Granting rights
User management commands, File quota and various file systems available, File System Management and Layout, File permissions, Login process, Managing Disk Quotas, Links (hard links, symbolic links)

- The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services.
- Consider, for example, that an operating system for a multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

Address Binding

- Memory management is the functionality of an operating system which handles or manages primary memory. Memory management keeps track of each and every memory location either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.
- Memory management provides protection by using two registers, a base register and a limit register. The base register holds the smallest legal physical memory address and the limit register specifies the size of the range. For example, if the base register holds 300000 and the limit register is 1209000, then the program can legally access all addresses from 300000 through 411999.

Instructions and data to memory addresses can be done in following ways

- Compile time -- When it is known at compile time where the process will reside, compile time binding is used to generate the absolute code.
- Load time -- When it is not known at compile time where the process will reside in memory, then the compiler generates re-locatable code.
- Execution time -- If the process can be moved during its execution from one memory segment to another, then binding must be delayed to be done at run time

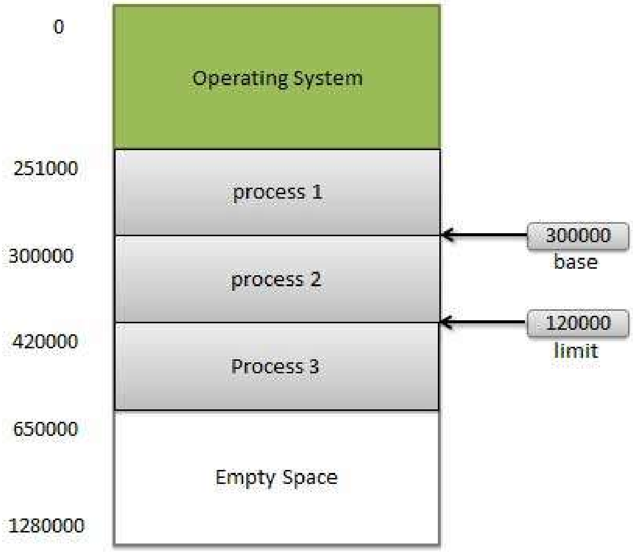
Dynamic Loading

- In dynamic loading, a routine of a program is not loaded until it is called by the program. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed. Other routines methods or modules are loaded on request. Dynamic loading makes better memory space utilization and unused routines are never loaded.
- The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

Dynamic Linking

- Linking is the process of collecting and combining various modules of code and data into a executable file that can be loaded into memory and executed. Operating system can link system level libraries to a program. When it combines the libraries at load time, the linking is called static linking and when this linking is done at the time of execution, it is called as dynamic linking.

- In static linking, libraries linked at compile time, so program code size becomes bigger whereas in dynamic linking libraries linked at execution time so program code size remains smaller.
- Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.



PHYSICAL AND VIRTUAL ADDRESS SPACE

Logical (Virtual) versus Physical Address Space

An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address. Logical address is also known a Virtual address. Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

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

MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

MEMORY ALLOCATION STRATEGIES

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.
- We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Protection

- Before discussing memory allocation further, we must discuss the issue of memory protection. If we have a system with a relocation register, together with a limit, we accomplish our goal. The relocation register contains the value of the smallest physical

address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).

- Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple **partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management.
- In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes.
- As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.

- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- In general, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes. This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

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

• **Best fit.** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

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

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation.** As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory

space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

- Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.
- One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. Two complementary techniques achieve this solution: segmentation and paging

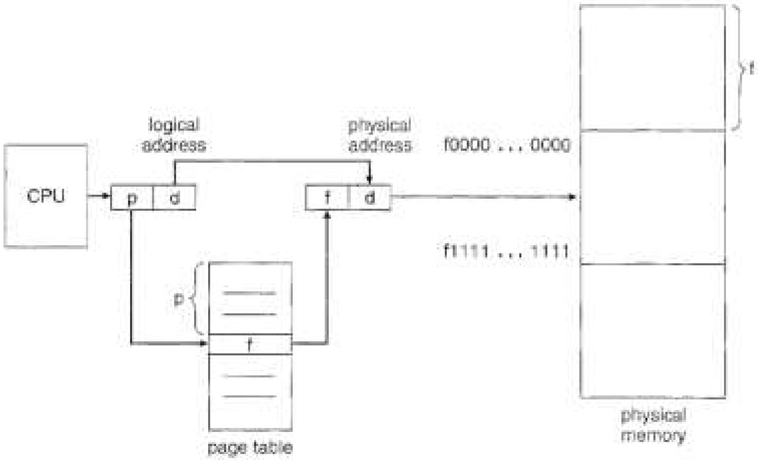
PAGING

- It is a memory-management scheme that permits the physical address space a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be framed on the backing store.
- The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems.

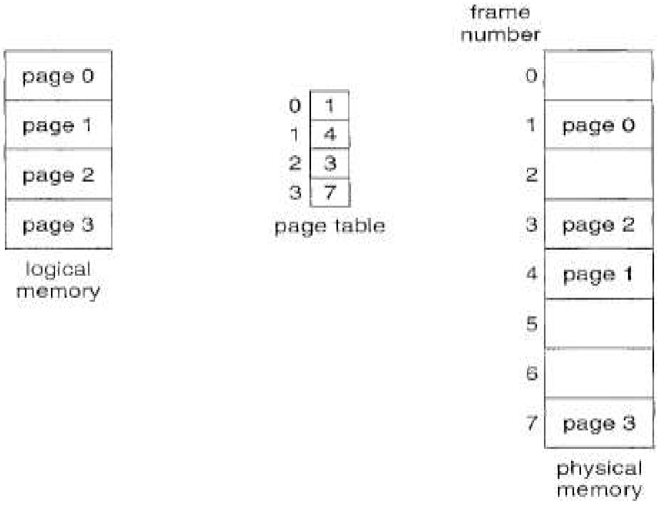
  Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

Basic Method

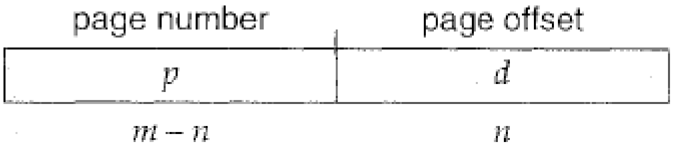
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are of the san1.e size as the memory frames. The hardware support for paging is illustrated in the following figure



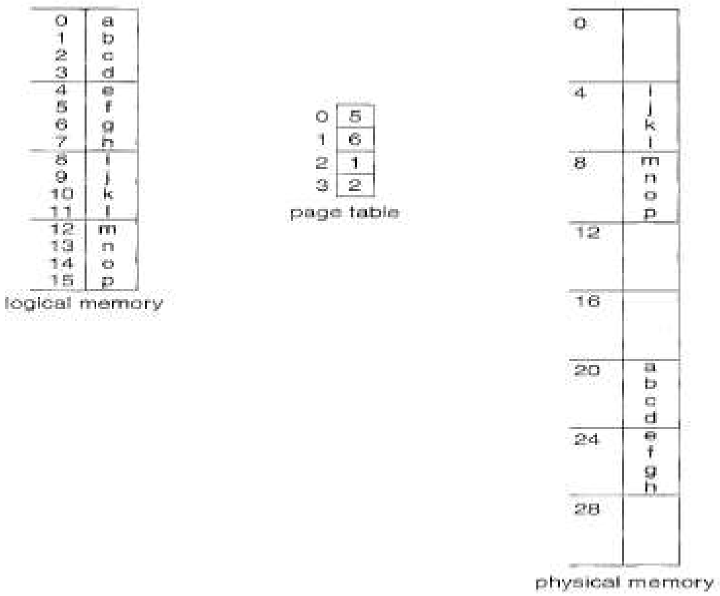
- Every address generated the CPU is divided into two parts: a {p) and a . The page number is used as an index into a page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in the following diagram



- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.



- If the size of the logical address space is 2m, and a page size is 271 addressing units (bytes or wordst then the high-order m- n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows: where p is an index into the page table and d is the displacement within the page. As a concrete (although minuscule) example, consider the memory in the following diagram



- Here, in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 x 4) + O]. Logical address 13 maps to physical address 9. You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address.

Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory. When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units.

- If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 - 1,086 = 962 bytes. In the worst case, a process would need 11 pages plus 1 byte. It would be allocated 11 + 1 frames, resulting in internal fragmentation of almost an entire frame. If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.
- Today, pages typically are between 4 KB and 8 KB in size and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages. Researchers are now developing support for variable on-the-fly page size. Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 232 physical page frames. If frame size is 4 KB, then a system with 4-byte entries can address 244 bytes (or 16 TB) of physical memory. When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires 11 pages, at least 11 frames must be available in memory.
- If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on. An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views

memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.

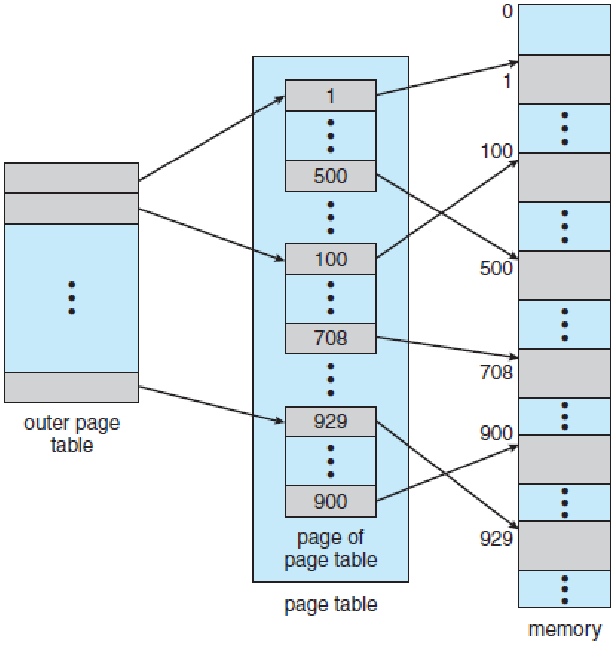
- The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own.
- It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns. Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory-which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame the frame-table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.
- In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses.If a user makes a system call (to do I/0, for example) and provides an address as a parameter (a buffe1~ for instance), that address must be mapped to produce the correct physical address.
- The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

STRUCTURE OF PAGE TABLE

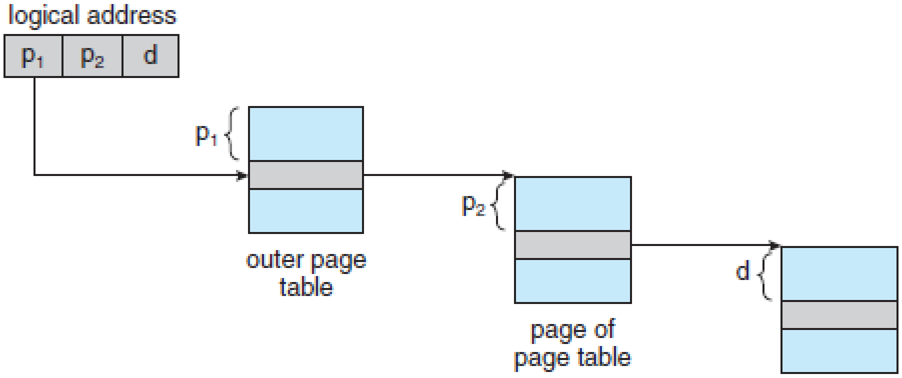
In this section, we explore some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

Hierarchical Paging

- Most modern computer systems support a large logical address space (232 to 264). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (212), then a page table may consist of up to 1 million entries (232/212). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces.



- We can accomplish this division in several ways. One way is to use a two-level paging algorithm, in which the page table itself is also paged. For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number

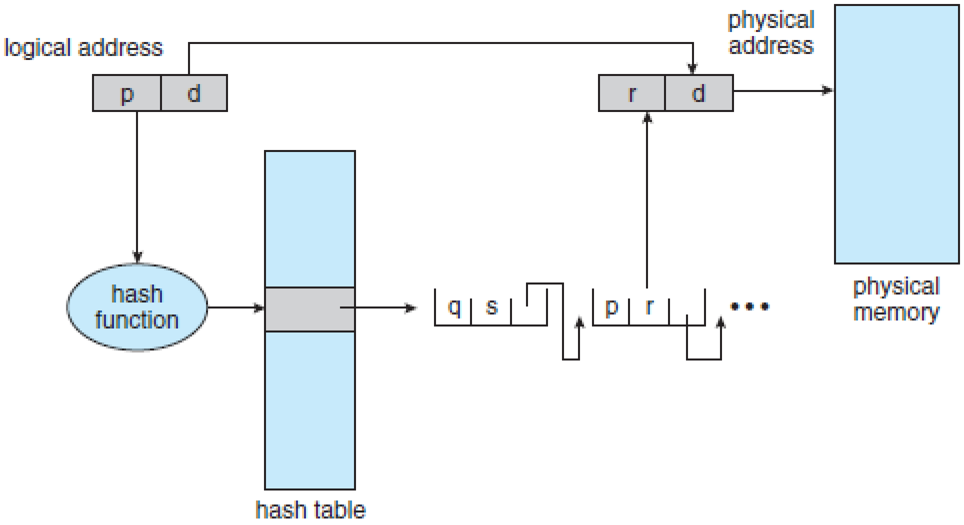


consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

Where p1 is an index into the outer page table and p2 is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure. Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

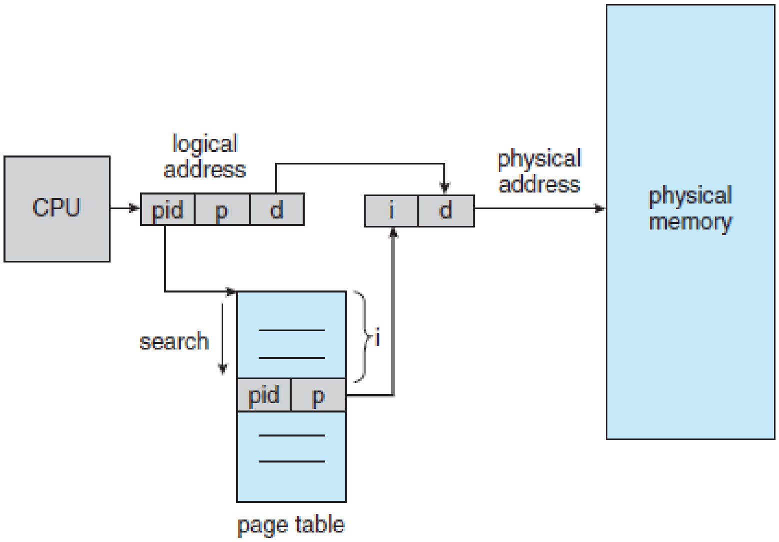
Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list. The algorithm works as follows:
- The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



Inverted Page Tables

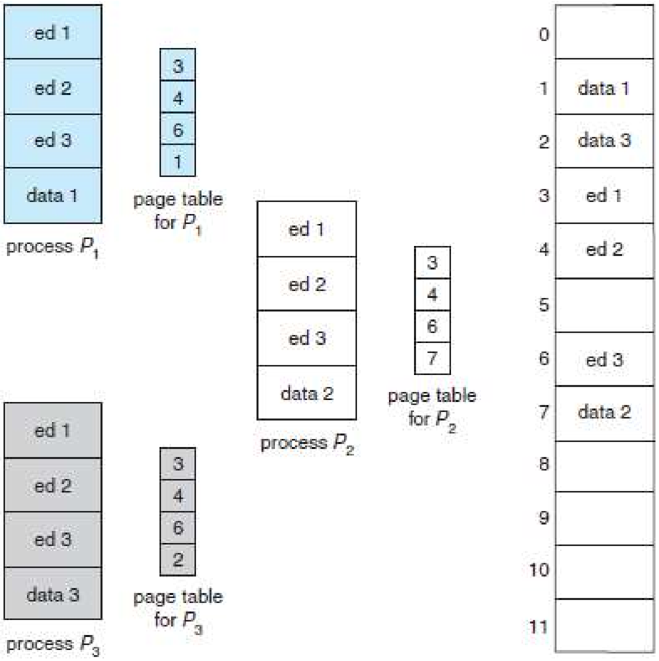
- Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter’s validity). This table representation is a natural one, since processes reference pages through the pages’ virtual addresses.
- The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.



- To solve this problem, we can use an inverted page table. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

<u>Shared Pages</u>

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code (or pure code), it can be shared, as shown in Figure. Here, we see three processes sharing a three-page editor—each page 50 KB in size (the large page size is used to simplify the figure). Each process has its own data page. Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.



- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings. Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.
- The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads.

SEGMENTATION

- An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory from the actual physical memory. As we have already seen, the user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. This mapping allows differentiation between logical memory and physical memory.

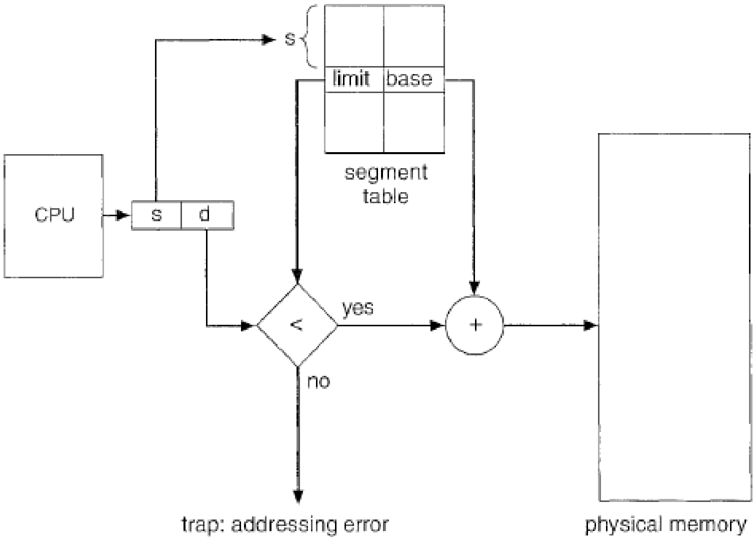
Basic Methods

- It is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.) For simplicity of implementation, segments are numbered and are referred to by a segn"lent number, rather than by a segment name. Thus, a logical address consists of a two tuple:

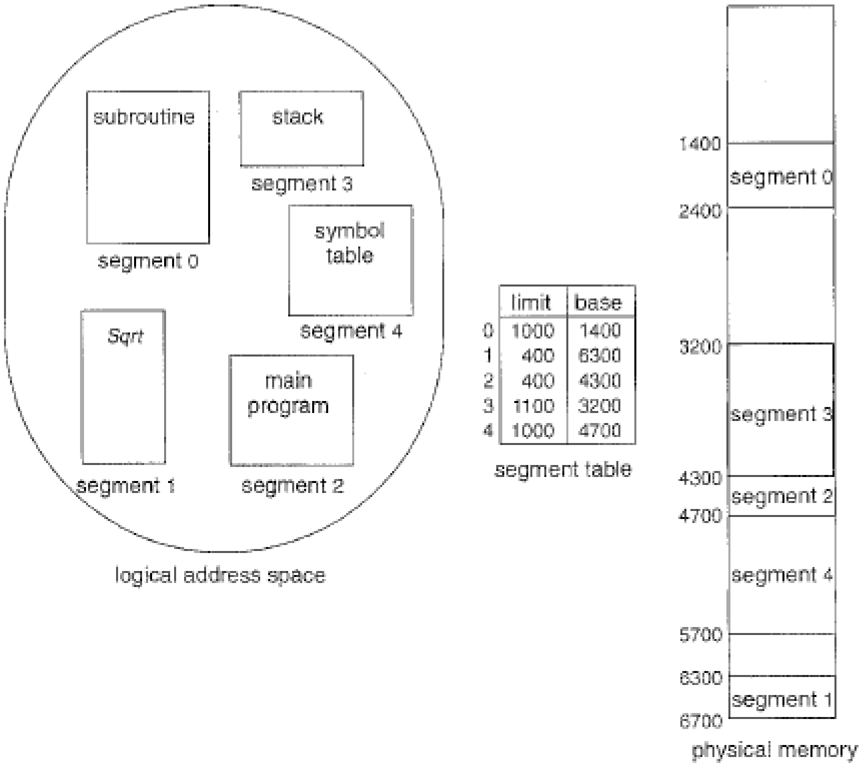
  <segment-number, offset>.
- Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A C compiler might create separate segments for the following:
  1. The code
  2. Global variables
  3. The heap, from which memory is allocated
  4. The stacks used by each thread
  5. The standard C library
- Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Hardware

- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by each entry in the segment table has a segment base and a segment limit. The segment base contains the start physical address where the segment resides in memory, and the segment limit specifies the length of the segment. The use of a segment table is illustrated in Figure



- A logical address consists of two parts: a segment number, s, and an offset into that segment, d. the segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs. As an example, consider the situation shown in Figure



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

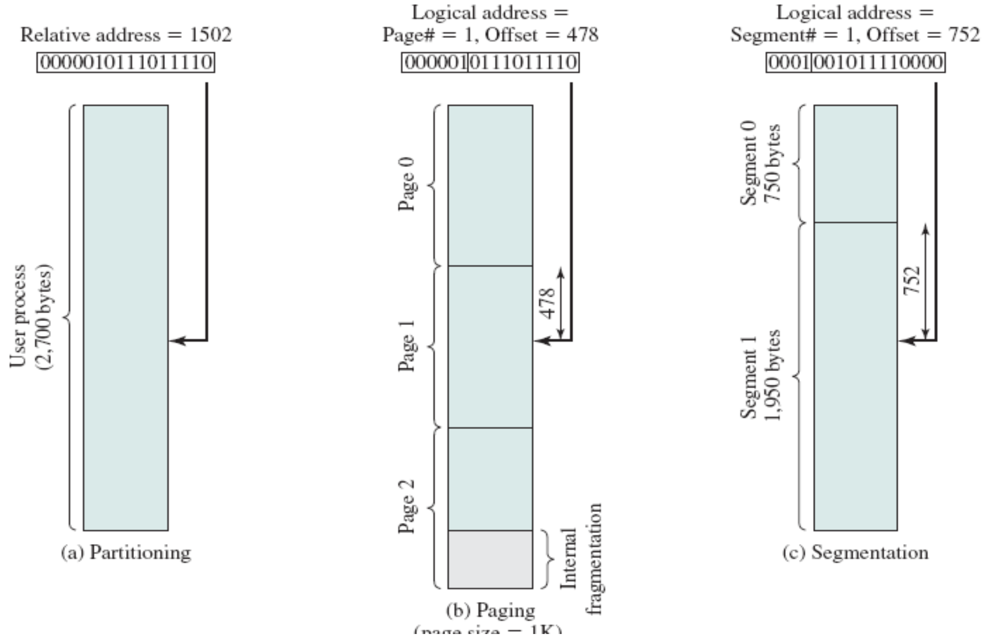
- We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

Segmentation and Paging

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

segment number and an offset. Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning.

- In the absence of an overlay scheme or the use of virtual memory, it would be required that all of a program's segments be loaded into memory for execution. The difference, compared to dynamic partitioning, is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation.
- However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less. Whereas paging is invisible to the programmer, segmentation is usually visible and is provided as a convenience for organizing programs and data. STypically, the programmer or compiler will assign programs and data to different segments. For purposes of modular programming, the program or data may be further broken down into multiple segments.
- The principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation. Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses.
- Each segment table entry would have to give the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment, to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware. Consider an address of n_m bits, where the leftmost n bits are the segment number and the rightmost m bits are the offset. In our example (Figure C), n _ 4 and m _ 12. Thus the maximum segment size is 2 12 _ 4096.



(a) Partitioning

(b) Paging (page size = 1K)

(c) Segmentation

The following steps are needed for address translation:

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

• Use the segment number as an index into the process segment table to find the starting physical address of the segment.

• Compare the offset, expressed in the rightmost m bits, to the length of the segment.

If the offset is greater than or equal to the length, the address is invalid. The desired physical address is the sum of the starting physical address of the segment plus the offset.

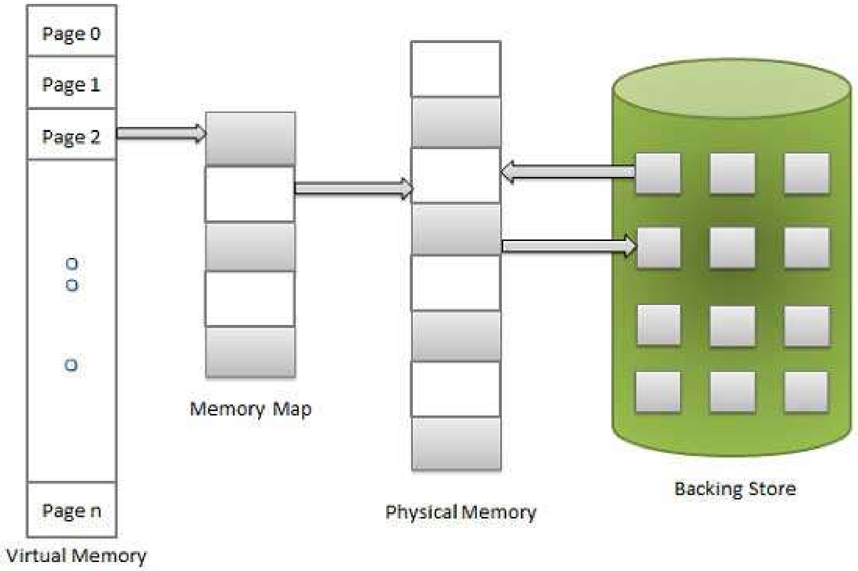
In our example, we have the logical address 0001001011110000, which is segment number 1, offset 752. Suppose that this segment is residing in main memory starting at physical address 0010000000100000. Then the physical address is 0010000000100000 + 001011110000 _ 0010001100010000.

To summarize, with simple segmentation, a process is divided into a number of segments that need not be of equal size. When a process is brought in, all of its segments are loaded into available regions of memory, and a segment table is set up.

VIRTUAL MEMORY

Virtual Memory and its Organization

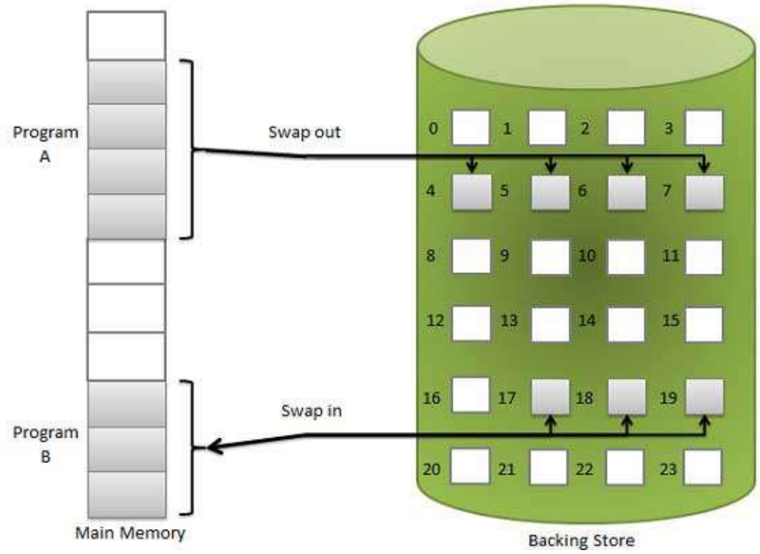
- Virtual memory is a technique that allows the execution of processes which are not completely available in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Following are the situations, when entire program is not required to be loaded fully in main memory.
- User written error handling routines are used only when an error occured in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.



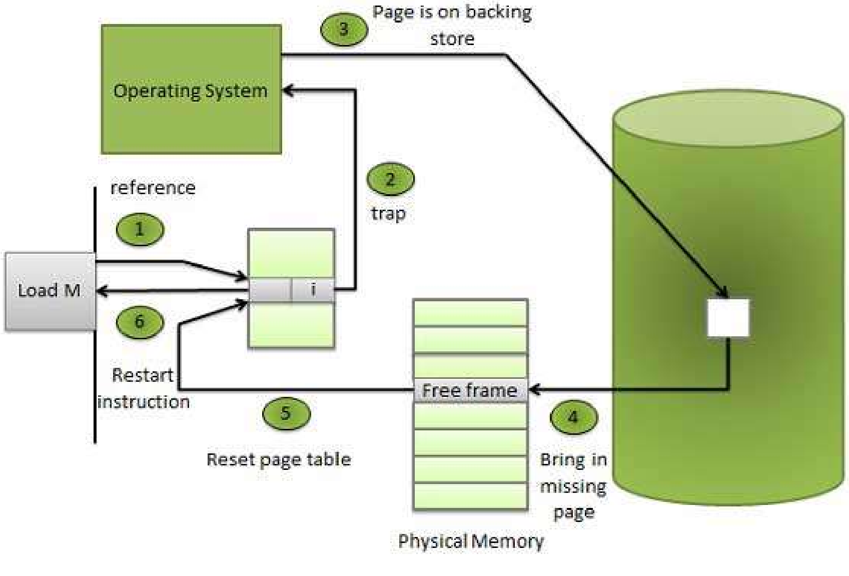
- Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

- A demand paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper called pager.
- When a process is to be swapped in, the pager, guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.
- Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme, where valid and invalid pages can be checked by checking the bit. Marking a page will have no effect if the process never attempts to access the page. While the process executes and accesses pages that are memory resident, execution proceeds normally.



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



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

➢ **Advantages**

Following are the advantages of Demand Paging

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

➢ **Disadvantages**

Following are the disadvantages of Demand Paging

- Number of tables and amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.
- Due to the lack of an explicit constraint on a job address space size.

POSSIBLE QUESTIONS

UNIT – III

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What do you mean by granting right?
2. What is user management commands?
3. What is File Quota?
4. Define File Layout.
5. What do you mean by managing disk quotas?
6. Define symbolic link.

PART – C (6 MARKS)

1. Explain the types of users.
2. Explain the user management commands.
3. Explain the file quota and various file systems available.
4. Discuss the file system management and layout.
5. Explain the file permissions.
6. Discuss the login process.

UNIT – III

SYLLABUS

User Management and the File System Types of Users, Creating users, Granting rights User management commands, File quota and various file systems available, File System Management and Layout, File permissions, Login process, Managing Disk Quotas, Links (hard links, symbolic links)

- The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services.
- Consider, for example, that an operating system for a multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers. This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

Address Binding

- Memory management is the functionality of an operating system which handles or manages primary memory. Memory management keeps track of each and every memory location either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.
- Memory management provides protection by using two registers, a base register and a limit register. The base register holds the smallest legal physical memory address and the limit register specifies the size of the range. For example, if the base register holds 300000 and the limit register is 1209000, then the program can legally access all addresses from 300000 through 411999.

Instructions and data to memory addresses can be done in following ways

- Compile time -- When it is known at compile time where the process will reside, compile time binding is used to generate the absolute code.
- Load time -- When it is not known at compile time where the process will reside in memory, then the compiler generates re-locatable code.
- Execution time -- If the process can be moved during its execution from one memory segment to another, then binding must be delayed to be done at run time

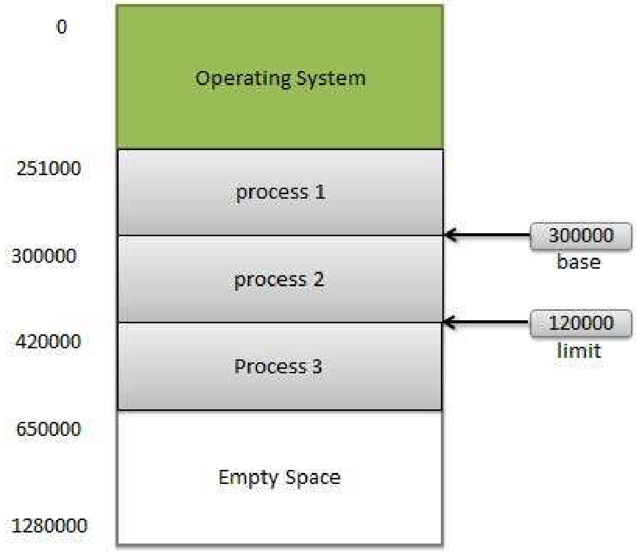
Dynamic Loading

- In dynamic loading, a routine of a program is not loaded until it is called by the program. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed. Other routines methods or modules are loaded on request. Dynamic loading makes better memory space utilization and unused routines are never loaded.
- The advantage of dynamic loading is that a routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- Dynamic loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

Dynamic Linking

- Linking is the process of collecting and combining various modules of code and data into a executable file that can be loaded into memory and executed. Operating system can link system level libraries to a program. When it combines the libraries at load time, the linking is called static linking and when this linking is done at the time of execution, it is called as dynamic linking.

- In static linking, libraries linked at compile time, so program code size becomes bigger whereas in dynamic linking libraries linked at execution time so program code size remains smaller.
- Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.



PHYSICAL AND VIRTUAL ADDRESS SPACE

Logical (Virtual) versus Physical Address Space

An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address. Logical address is also known a Virtual address. Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

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

MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

MEMORY ALLOCATION STRATEGIES

Contiguous Memory Allocation

- The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.
- We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Protection

- Before discussing memory allocation further, we must discuss the issue of memory protection. If we have a system with a relocation register, together with a limit, we accomplish our goal. The relocation register contains the value of the smallest physical

address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).

- Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple **partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management.
- In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes.
- As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.

- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.
- In general, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes. This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

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

• **Best fit.** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

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

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation.** As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory

space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

- Whether we are using the first-fit or best-fit strategy can affect the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece—the one on the top or the one on the bottom?) Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself.
- The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation**—unused memory that is internal to a partition.
- One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. Two complementary techniques achieve this solution: segmentation and paging

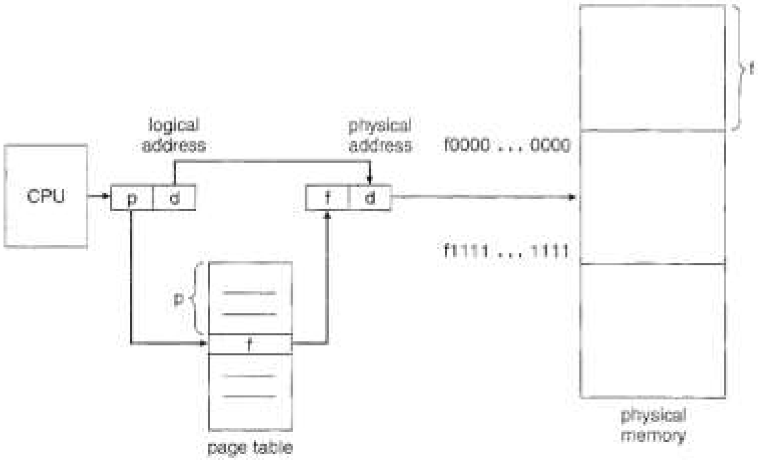
PAGING

- It is a memory-management scheme that permits the physical address space a process to be noncontiguous. Paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store; most memory management schemes used before the introduction of paging suffered from this problem. The problem arises because, when some code fragments or data residing in main memory need to be swapped out, space must be framed on the backing store.
- The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems.

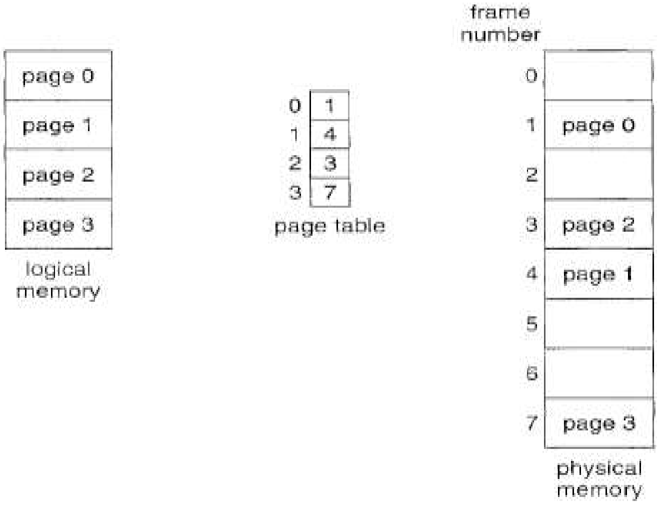
  Traditionally, support for paging has been handled by hardware. However, recent designs have implemented paging by closely integrating the hardware and operating system, especially on 64-bit microprocessors.

Basic Method

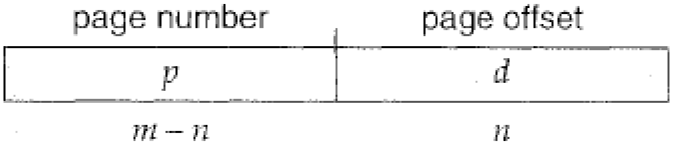
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are of the san1.e size as the memory frames. The hardware support for paging is illustrated in the following figure



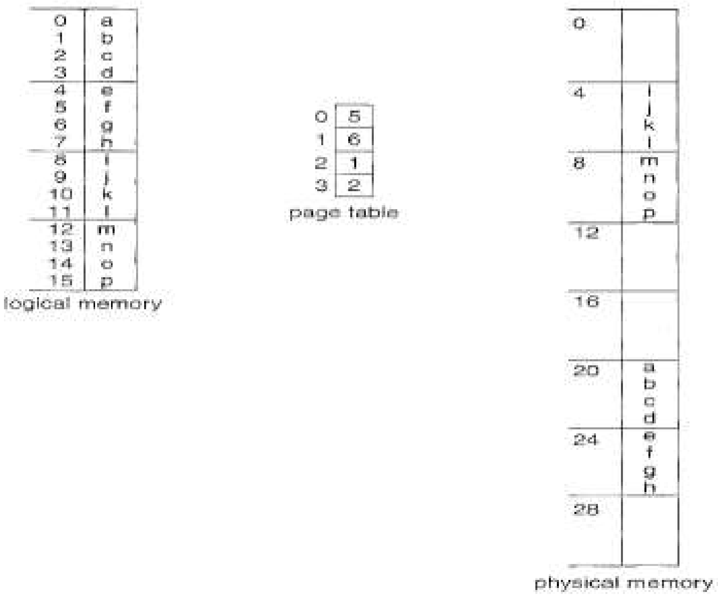
- Every address generated the CPU is divided into two parts: a {p) and a . The page number is used as an index into a page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in the following diagram



- The page size (like the frame size) is defined by the hardware. The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.



- If the size of the logical address space is 2m, and a page size is 271 addressing units (bytes or wordst then the high-order m- n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows: where p is an index into the page table and d is the displacement within the page. As a concrete (although minuscule) example, consider the memory in the following diagram



- Here, in the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the user's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 x 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 x 4) + O]. Logical address 13 maps to physical address 9. You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address.

Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory. When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units.

- If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 - 1,086 = 962 bytes. In the worst case, a process would need 11 pages plus 1 byte. It would be allocated 11 + 1 frames, resulting in internal fragmentation of almost an entire frame. If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.
- Today, pages typically are between 4 KB and 8 KB in size and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages. Researchers are now developing support for variable on-the-fly page size. Usually, each page-table entry is 4 bytes long, but that size can vary as well. A 32-bit entry can point to one of 232 physical page frames. If frame size is 4 KB, then a system with 4-byte entries can address 244 bytes (or 16 TB) of physical memory. When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires 11 pages, at least 11 frames must be available in memory.
- If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on. An important aspect of paging is the clear separation between the user's view of memory and the actual physical memory. The user program views

memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs.

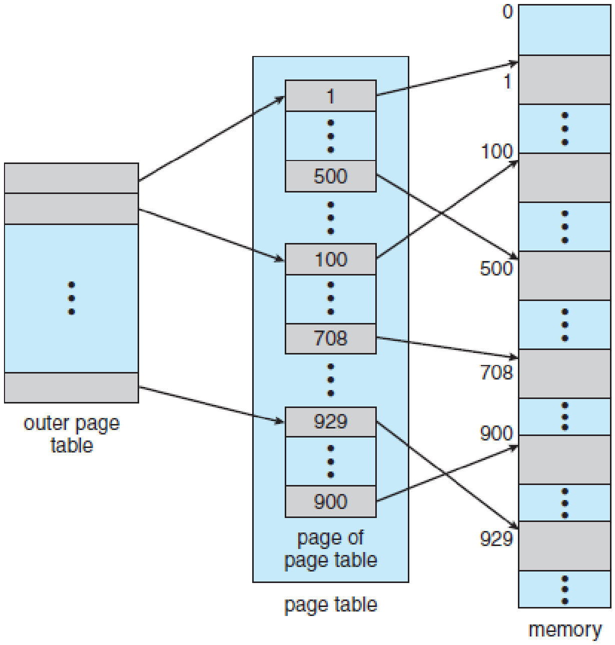
- The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the user and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own.
- It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns. Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory-which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame the frame-table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.
- In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses.If a user makes a system call (to do I/0, for example) and provides an address as a parameter (a buffe1~ for instance), that address must be mapped to produce the correct physical address.
- The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

STRUCTURE OF PAGE TABLE

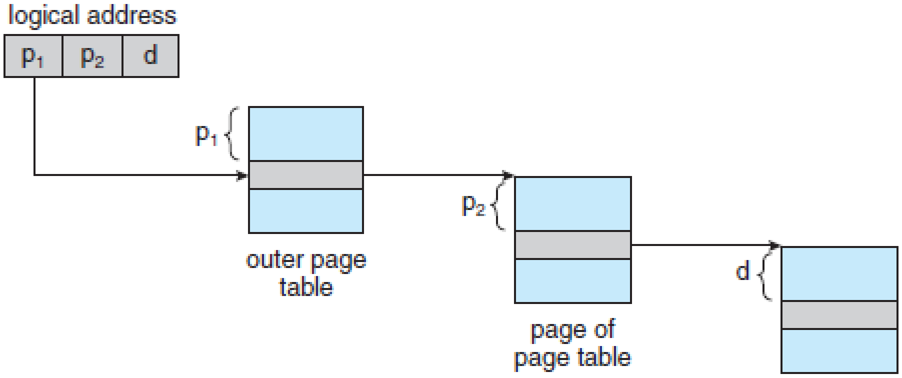
In this section, we explore some of the most common techniques for structuring the page table, including hierarchical paging, hashed page tables, and inverted page tables.

Hierarchical Paging

- Most modern computer systems support a large logical address space (232 to 264). In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (212), then a page table may consist of up to 1 million entries (232/212). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces.



- We can accomplish this division in several ways. One way is to use a two-level paging algorithm, in which the page table itself is also paged. For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number

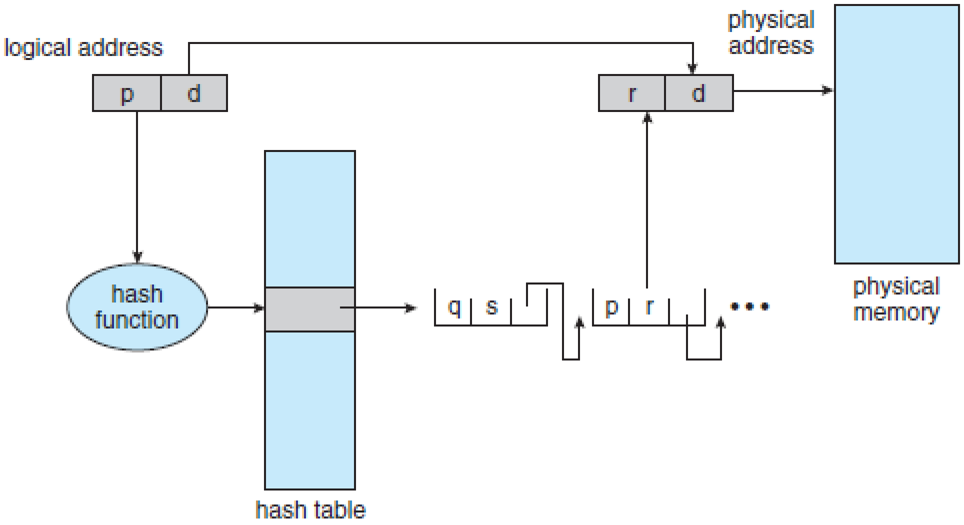


consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

Where p1 is an index into the outer page table and p2 is the displacement within the page of the inner page table. The address-translation method for this architecture is shown in Figure. Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

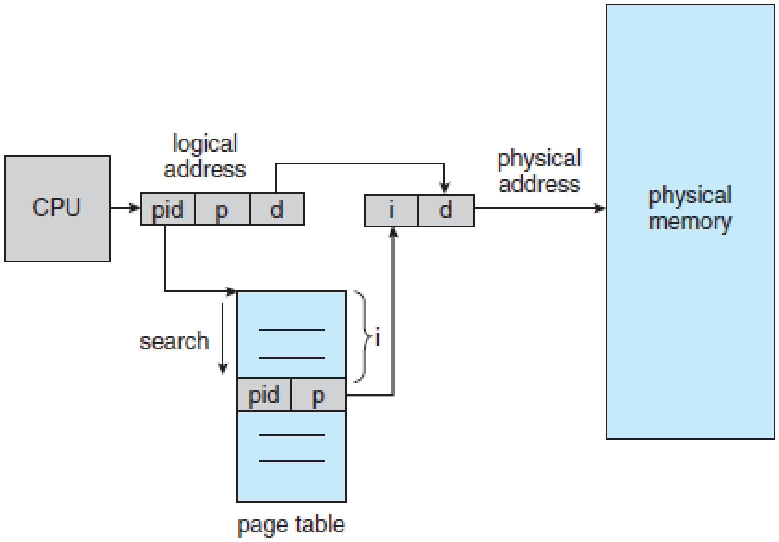
Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list. The algorithm works as follows:
- The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



Inverted Page Tables

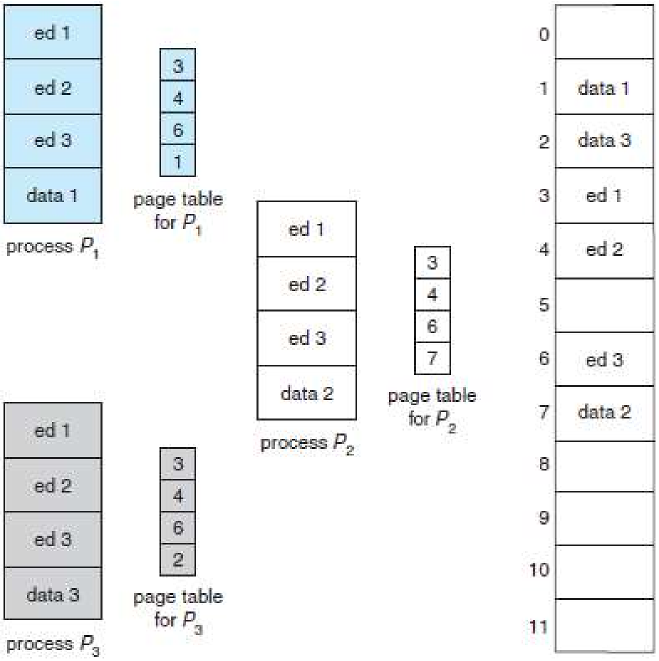
- Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses.
- The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.



- To solve this problem, we can use an inverted page table. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

Shared Pages

- An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment. Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. If the code is reentrant code (or pure code), it can be shared, as shown in Figure. Here, we see three processes sharing a three-page editor—each page 50 KB in size (the large page size is used to simplify the figure). Each process has its own data page. Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time.



- Each process has its own copy of registers and data storage to hold the data for the process's execution. The data for two different processes will, of course, be different. Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames. Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings. Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on. To be sharable, the code must be reentrant. The read-only nature of shared code should not be left to the correctness of the code; the operating system should enforce this property.
- The sharing of memory among processes on a system is similar to the sharing of the address space of a task by threads.

SEGMENTATION

- An important aspect of memory management that became unavoidable with paging is the separation of the user's view of memory from the actual physical memory. As we have already seen, the user's view of memory is not the same as the actual physical memory. The user's view is mapped onto physical memory. This mapping allows differentiation between logical memory and physical memory.

Basic Methods

- It is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. (Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.) For simplicity of implementation, segments are numbered and are referred to by a segn"lent number, rather than by a segment name. Thus, a logical address consists of a two tuple:

  <segment-number, offset>.

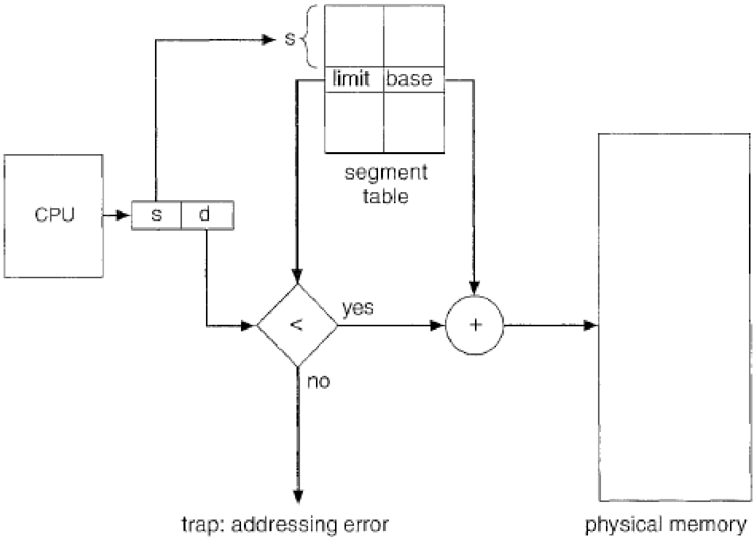
- Normally, the user program is compiled, and the compiler automatically constructs segments reflecting the input program. A C compiler might create separate segments for the following:

  1. The code
  2. Global variables
  3. The heap, from which memory is allocated
  4. The stacks used by each thread
  5. The standard C library

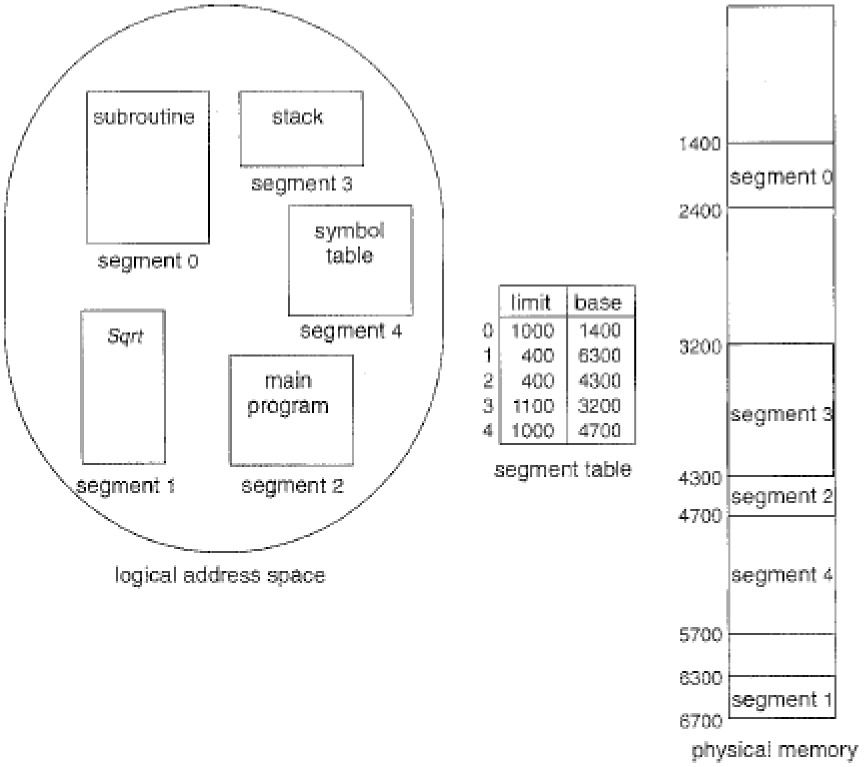
- Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Hardware

- Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes. Thus, we must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected by each entry in the segment table has a segment base and a segment limit. The segment base contains the start physical address where the segment resides in memory, and the segment limit specifies the length of the segment. The use of a segment table is illustrated in Figure



- A logical address consists of two parts: a segment number, s, and an offset into that segment, d. the segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs. As an example, consider the situation shown in Figure



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

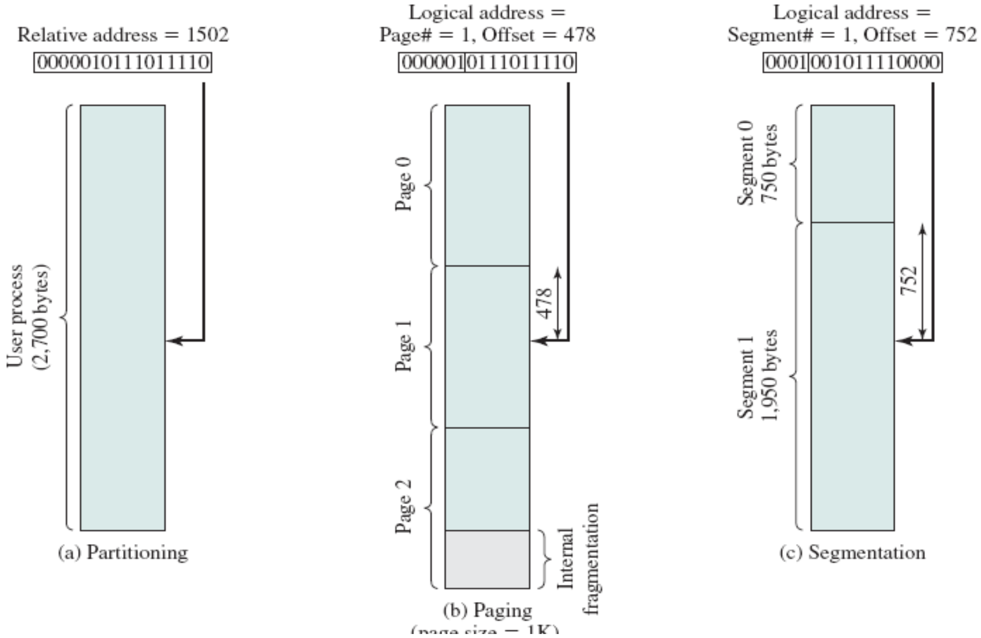
- We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1000 bytes long.

Segmentation and Paging

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

segment number and an offset. Because of the use of unequal-size segments, segmentation is similar to dynamic partitioning.

- In the absence of an overlay scheme or the use of virtual memory, it would be required that all of a program's segments be loaded into memory for execution. The difference, compared to dynamic partitioning, is that with segmentation a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation.
- However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less. Whereas paging is invisible to the programmer, segmentation is usually visible and is provided as a convenience for organizing programs and data. STypically, the programmer or compiler will assign programs and data to different segments. For purposes of modular programming, the program or data may be further broken down into multiple segments.
- The principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation. Another consequence of unequal-size segments is that there is no simple relationship between logical addresses and physical addresses.
- Each segment table entry would have to give the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment, to assure that invalid addresses are not used. When a process enters the Running state, the address of its segment table is loaded into a special register used by the memory management hardware. Consider an address of n_m bits, where the leftmost n bits are the segment number and the rightmost m bits are the offset. In our example (Figure C), n _ 4 and m _ 12. Thus the maximum segment size is 2 12 _ 4096.



The following steps are needed for address translation:

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

• Use the segment number as an index into the process segment table to find the starting physical address of the segment.

• Compare the offset, expressed in the rightmost m bits, to the length of the segment.

If the offset is greater than or equal to the length, the address is invalid. The desired physical address is the sum of the starting physical address of the segment plus the offset.

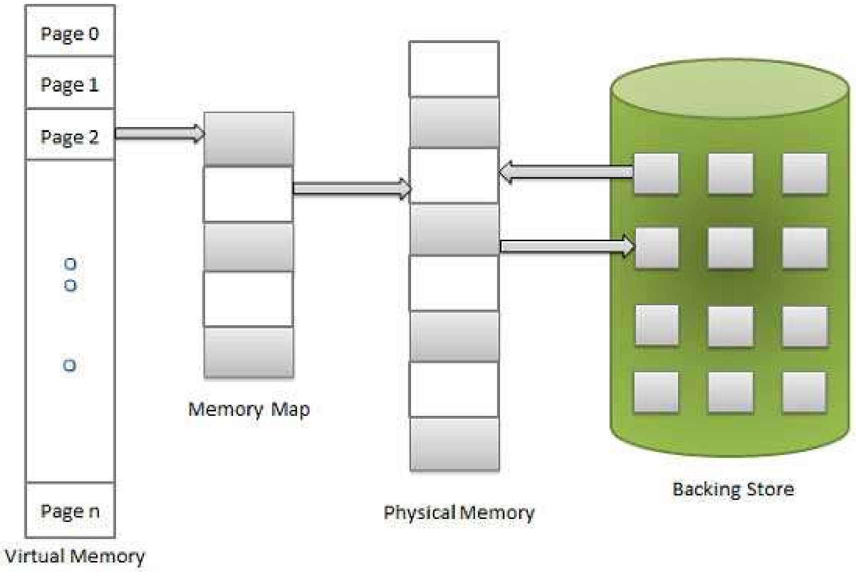
In our example, we have the logical address 0001001011110000, which is segment number 1, offset 752. Suppose that this segment is residing in main memory starting at physical address 0010000000100000. Then the physical address is 0010000000100000 + 001011110000 _ 0010001100010000.

To summarize, with simple segmentation, a process is divided into a number of segments that need not be of equal size. When a process is brought in, all of its segments are loaded into available regions of memory, and a segment table is set up.

VIRTUAL MEMORY

Virtual Memory and its Organization

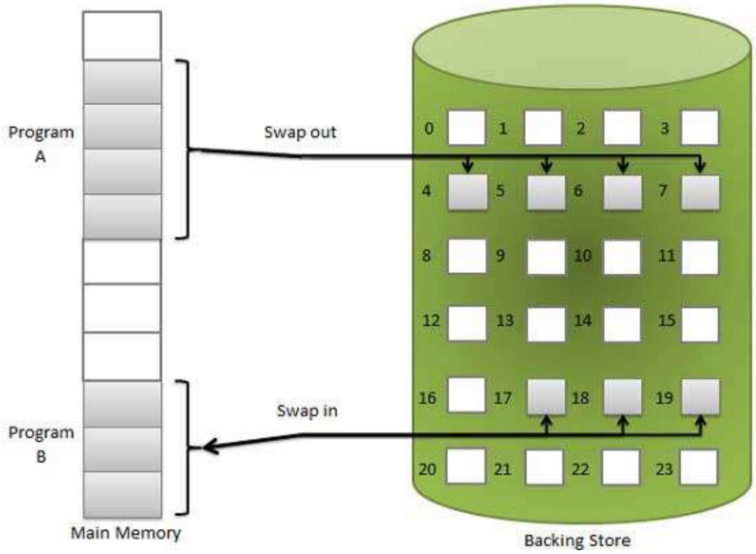
- Virtual memory is a technique that allows the execution of processes which are not completely available in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Following are the situations, when entire program is not required to be loaded fully in main memory.
- User written error handling routines are used only when an error occured in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.



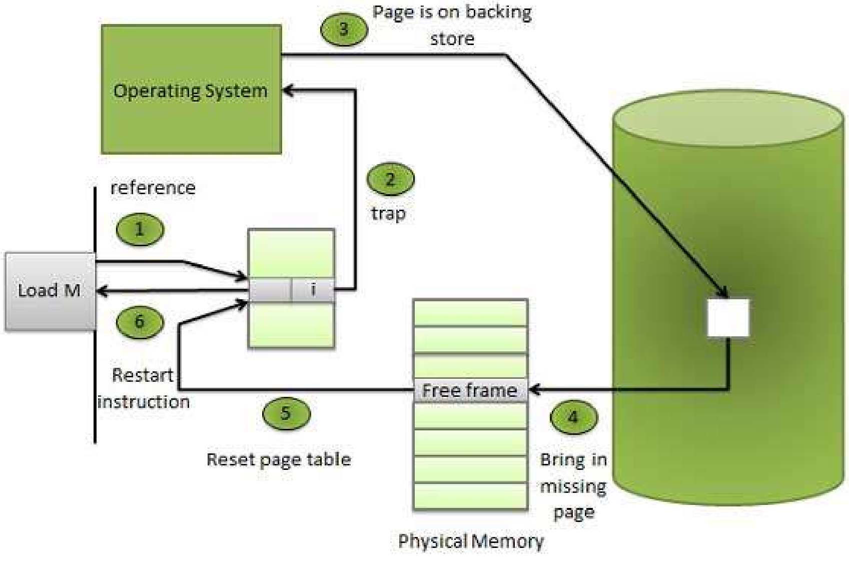
- Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

- A demand paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper called pager.
- When a process is to be swapped in, the pager, guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.
- Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme, where valid and invalid pages can be checked by checking the bit. Marking a page will have no effect if the process never attempts to access the page. While the process executes and accesses pages that are memory resident, execution proceeds normally.



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



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

➢ **Advantages**

Following are the advantages of Demand Paging

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

- **Disadvantages**

Following are the disadvantages of Demand Paging

- Number of tables and amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.
- Due to the lack of an explicit constraint on a job address space size.

POSSIBLE QUESTIONS

UNIT – III

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What do you mean by granting right?
2. What is user management commands?
3. What is File Quota?
4. Define File Layout.
5. What do you mean by managing disk quotas?
6. Define symbolic link.

PART – C (6 MARKS)

1. Explain the types of users.
2. Explain the user management commands.
3. Explain the file quota and various file systems available.
4. Discuss the file system management and layout.
5. Explain the file permissions.
6. Discuss the login process.

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - IV : (Objective Type Multiple choice Questions each Question carries one Mark)

OPERATING SYSTEMS

PART - A (Online Examination)

Questions	Opt1	opt2	opt3	opt4	KEY
The file system consist of -------------- Distinct parts	2	3	4	5	2
A --------------- File is a sequence of character organized into lines	Source	Object	Text	Executable	Text
A --------------- File is a sequence of subroutines and functions	Source	Object	Text	Executable	Source
The operating system keeps a small table called the ---------------- ,containing information about all open files	Show file table	Visible file table	Open file ta	Manage file table	Open file table
A file is executed in --------------- extension	External structure	.bat	.mdb	.in	.bat
The .bat file is a ----------------containing in ANCII format,command to the operating system	Binary file	Batch file	Text file	Word file	Batch file
The file type is used to indicate the ---------------- of the file	.txt	Internal structure	Block structure	Outer structure	Internal structure
Information in the file is processed in the order called------------------	Direct access	Sequence access	Dynamic access	Random access	Sequence access
A file is made up of fixed length that allows the program to read and write record rapidly in no	Direct access	Sequence access	Dyanamic access	Random access	Direct access

Data cannot be written in secondary storage unless written with in a ---------------------	File	Swap space	Directory	Text format	File
File attribute consist of ----------------	Name,Type,Content	Name,type,Size	Seperate directory system	Name,identifier	Name,Size,Type,identifier
The information about all files is kept in -----------------	swap space	operating system	Name,Size,Type,identifier	Hard disk	Seperate directory
A file is a -------------- type	Abstract	Primitive	Public	Private	Abstract
In UNIX Open system call returns -----------------	pointer to the entry in the open	pointer to the entry in the	A file to the process calling it	pointer to the entry in	pointer to the entry in the
The open file table has a ------------------- Associated with each file	File content	File permission	open count	Close count	open count
The file name is generaly split into which of the two parts -----------------	Name and type	Name and identifier	Name and extension	Extension and type	Name and extension
In the sequential access method, information in the file is processed	One disk after the other	One record after the other	One text document after	One name after the	One record after the
Sequential access method ----------------,on random access devices	Works well	Dosen’t works well	Works slow	Works normal	Works well
The direct access method is based on a ------------- model of a file as---------------- allow random access to	Magnetic tape,magnetic	Tape,Tapes	Disk,Disks	Tape,Disk	Disk,Disks
A relative block number is an index relative to ---------------------	The beginning of the file	The end of the file	The last written position in file	Middle of the file	The beginning of the file
The index contains ------------------------	Name of all content of file	Pointer to each page	Pointers to the various blocks	Pointer to same page	Pointers to the various
The directory can be viewed as a ---------------,that translate the file name into their directory entries	Symbol table	Partition	Swap space	Cache	Symbol table
In the single level directory: ------------------------	All files are contain in	All files are contained in	Depend on the operating system	Depend on the file name	All files are contained in
In the single level directory --------------	All directory must have a unique	All files must have a unique	All files must have a unique owner	All files must have a	All files must have a unique
In the two level directory structure -----------------	own user file directory	has its own master file	)Both a and b	its different file directory	Both a and b

When a user refers to a particular file --------------------	System MFD is searched	His own UFD is searched	Both MFD and UFD are searched	directory is searched	UFD are searched
The disadvantage of the two level directory structure is that	the name collision problem	name collision problem	users from one another	users from one another	users from one another
In the tree structure directory -------------------	The tree has the same directory	The tree has the leaf	The tree has the root directory	The tree has no directory	The tree has the root
The three major methods of allocating disk space that are in wide use are ---------------	Contiguous,Linked ,Hashed	Contiguous,Linked,Indexed	Linked,Hashed,Indexed	Contiguous ,Linked	Contiguous,Linked,Indexed
In Contiguous allocation ----------------	occupy a set of contiguous block	linked list of disk blocks	All the pointers to scattered	All the files are blocked	occupy a set of contiguous
In linked allocation ------------------	Each file must occupy a set of	Each file is a linked list of	All the pointers to scattered	All the files are blocked	Each file is a linked list of
In indexed allocation ------------	Each file must occupy a set of	Each file is a linked list of	All the pointers to scattered blocks	All the files are blocked	All the pointers to
One system where there are multiple operating system, the decision to load a particular one is done by ----------------	Boot loader	Boot strap	Process control block	File control block	Boot loader
The VFS refers to -----------	Virtual File System	Valid File System	Virtual Font System	Virtual Function	Virtual File System
The disadvantage of a linear list of directory entries is the ---------------------	Size of the linear list in the memory	Linear search to find a file	It is not reliable	It is not valid	Linear search to find a file
One difficulty of contiguous allocation is ---------------	Finding space for a new file	Ineffecient	Costly	Time taking	Finding space for a new file
To solve the problem of external fragmentation ----------------- needs to be done periodically	Compaction	Check	Formatting	Replacing memory	Compaction
If too little space is allocated to a file ----------------	The file will not work	There will not be any space	The file cannot be extended	file cannot be opened	cannot be extended
A system program such as fsck ------------------ is a consistency checker	UNIX	Windows	Macintosh	Solaris	UNIX
Each set of operations for performing a specific task is a ----------------------	Program	Code	Transaction	Method	Transaction

Once the changes are written to the log, they are considered to be ---------------	Committed	Aborted	Completed	Finished	Committed
When an entire command transaction is completed,-------------------	It is stored in the memory	from the log file	It is redone	from the memory	from the log file
In --------------- information is recorded magnetically on platters	Magnetic disk	Electrical disk	Assemblies	Cylinders	Magnetic disk
The head of the magnetic disk are attached to a --------------- that moves all the head as unit	Spindle	Disk arm	Track	Pointer	Disk arm
The set of tracks that are at one arm position make up a -----------	Magnetic disk	Electrical disk	Assemblies	Cylinders	Cylinders
The time taken to move a disk arm to the desired cylinder is called as---------------	Positioning time	Random access ti	Seek time	Rotational latency	Seek time
When a head damages the magnetic surface, it is known as ---------------------	Disk crash	Head crash	Magnetic damage	All of these	Head crash
A flopy disk is designed to rotate -------------- as compared to a hard disk drive	Faster	Slower	At the same speed	Normal speed	Slower
The host controller is -------------------	the end of each disk	the computer end of the bus	Both a and b	the system side	the computer end of the bus
The process of dividing a disk into sectors that the disk controller can read and write, before a disk can store data is known as-----------------	Partitioning	Swap space creation	Low-level formatting	Physical formatting	Low-level formatting ,Physical
the data structure for a sector typically contains -------------------	Header	Data area	Trailer	Main section	Header ,Data area ,Trailer
The header and trailer of a sector contains information used by the disk controller such as ______________.	Main section	Error corecting codes	Sector number	Disk identifier	Sector number
The two steps that the operating system takes to use a disk to hold its files are ---------------- and --------------	partitioning	Swap space creation	Catching	Logical formatting	partitioning
The -------------- program initializes all aspects of the system, from CPU registers to device controllers and the content of main memory, and then starts the	Main	Boot loader	Boot strap	ROM	Boot strap

For most computers the boot strap is stored in--------------------	RAM	ROM	Cache	Tertiary storage	ROM
A disk that has a boot partition is called a ---------------	Start disk	Destroyed blocks	Boot disk	Format disk	System disk,boot disk
Defective sectors on disks are often known as -----------------	Good blocks	System disk	Bad blocks	Semi blocks	Bad blocks
Bad blocks are called as ___________	Good Sectors	Defective Sectors	boot disks	boot strap	Defective Sectors
ROM got __________ file	boot strap	Data area	head data	random data	boot strap

UNIT V

System calls, Using system calls Pipes and Filters, Decision making in Shell Scripts (If else, switch), Loops in shell, Functions, Utility programs (cut, paste, join, tr, uniq utilities), Pattern matching utility (grep)

Linux looks and feels much like any other UNIX system. Its development began in 1991, when a Finnish student, Linus Torvalds, wrote and christened **Linux, a** small but self-contained kernel for the 80386 processor, the first true 32-bit processor in Intel's range of PC-compatible CPUs. From an initial kernel that partially implemented a small subset of the UNIX system services, the Linux system has grown to include much ifFNIX functionality. In its early days, Linux development revolved largely around the central operating-system kernel—the core, privileged executive that manages all system resources and that interacts directly with the computer hardware. The **Linux kernel** is an entirely original piece of software developed from scratch by the Linux community. The **Linux system,** as we know it today, includes a multitude of components, some written from scratch, others borrowed from other development projects, and still others created in collaboration with other teams. A **Linux distribution** includes all the standard components of the Linux system, plus a set of administrative tools to simplify the initial installation and subsequent upgrading of Linux and to manage installation and removal of other packages on the system.

1.2 The Linux Kernel

The first Linux kernel released to the public was Version 0.01, dated May 14, 1991. It had no networking, ran only on 80386-compatible Intel processors and PC hardware, and had extremely limited device-driver support. The virtual memory subsystem was also fairly basic and included no support for memory mapped files; however, even this early incarnation supported shared pages with copy-on-write. The only file system supported was the Minix file system -the first Linux kernels were cross-developed on a Minix platform.

The next milestone version, Linux 1.0, was released on March 14, 1994. This release culminated three years of rapid development of the LimlX kernel. Perhaps the single biggest new feature was networking: 1.0 included support for UNIX's standard

TCP liP networking protocols, as well as a BSD-compatible socket interface for networking programming. Device-driver support was added for running IP over an Ethernet or (using PPP or SLIP protocols) over serial lines or modems. The 1.0 kernel also included a new, much enhanced file system without the limitations of the original Minix file system and supported a range of SCSI controllers for high-performance disk access. The developers extended the virtual memory subsystem to support paging to swap files and memory mapping of arbitrary files (but only read-only memory mapping was implemented in 1.0). A range of extra hardware support was also included in this release.

Although still restricted to the Intel PC platform, hardware support had grown to include floppy-disk and CD-ROM devices, as well as sound cards, a range of mice, and international keyboards. Floating-point emulation was provided in the kernel for 80386 users who had no 80387 math coprocessor; System V UNIX-style inclLlding shared memory, semaphores, and message queues, was implemented. Simple support for dynamically loadable and unloadable kernel modules was supplied as well. At this point, development started on the 1.1 kernel stream, but numerous bug-fix patches were released subsequently against 1.0.

In March 1995, the 1.2 kernel was released. This release did not offer nearly the same improvement in functionality as the 1.0 release, but it did support a much wider variety of hardware, including the new PCI hardware bus architecture. Developers added another PC-specific feature-support for the 80386 CPU's virtual8086 mode-to allow emulation of the DOS operating system for PC computers. They also updated the networking stack to provide support for the IPX protocol and made the IP implementation more complete by including accounting and firewalling functionality.

The 1.2 kernel was the final PC-only Linux kernel. The source distribution for Linux 1.2 included partially implemented support for SPARC, Alpha, and MIPS CPUs, but full integration of these other architectures did not begin until after the 1.2 stable kernels was released. The Linux 1.2 release concentrated on wider hardware support and more complete implementations of existing functionality.

Much new functionality was under development at the time, but integration of the new code into the main kernel source code had been deferred until after the stable 1.2 kernel had been released. As a result, the 1.3 development stream saw a great deal of new functionality added to the kernel. This work was finally released as Linux 2.0 in since 1996. This release was given a major version-number increment on account of two major new capabilities: support for multiple architectures, including a 64-bit native Alpha port, and support for multiprocessor architectures. Linux distributions based on 2.0 are also available for the Motorola 68000-series processors and for Sun's SPARC systems.

A derived version of Linux running on top of the Mach microkernel also runs on PC and PowerMac systems. The changes in 2.0 did not stop there. The memory-management code was substantially improved to provide a unified cache for file-system data independent of the caching of block devices. As a result of this change, the kernel offered greatly increased file-system and virtual memory performance. For the first time, file-system caching was extended to networked file systems, and writable memory-mapped regions also were supported. The 2.0 kernel also included much improved TCP /IP performance, and a number of new networking protocols were added, including Apple

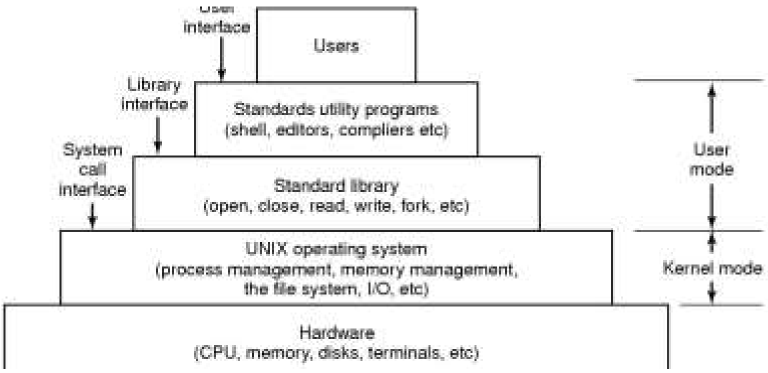
Talk, AX.25an'lateur radio networking, and ISDN support. The ability to mount remote netware and SMB (Microsoft LanManager) network volumes was added. Other major improvements in 2.0 were support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand. Dynamic configuration of the kernel at run time was much improved through a new, standardized configuration interface. Additional new features included file-system quotas and POSIX-compatible real-time process-scheduling classes. Improvements continued with the release of Linux 2.2 in January 1999.

A port for UltraSPARC systems was added. Networking was enhanced with more flexible firewalling, better routing and traffic management, and support for TCP large window and selective acks. Acorn, Apple, and NT disks could now be read, and NFS was enhanced and a kernel-mode NFS daemon added. Signal handling, interrupts, and some I/0 were locked at a finer level than before to improve symmetric multiprocessor (SMP) performance. Advances in the 2.4 and 2.6 releases of the kernel include increased support for SMP systems, journaling file systems, and enhancements to the memory management system. The process scheduler was modified in Version 2.6, providing an efficient 0(1) scheduling algorithm. In addition, the LimiX 2.6 kernel is now preemptive, allowing a process to be preempted while running in kernel mode.

1.3 The Linux System: Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools.
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.
- Main design goals are speed, efficiency, flexibility and standardization.
- Linux is designed to be compliant with the relevant POSIX documents; some Linux distributions have achieved official POSIX certification.
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior.

1.4 The layers of a UNIX system



User interface
Users
Library interface
Standards utility programs (shell, editors, compliers etc)
System call interface
Standard library (open, close, read, write, fork, etc)
UNIX operating system (process management, memory management, the file system, I/O, etc)
Hardware (CPU, memory, disks, terminals, etc)
User mode
Kernel mode

1.5 Components of a UNIX System

- Like most UNIX implementations, Linux is composed of three main bodies of code:
  - Standard Utilities Programs
  - Standard Library
  - Kernel
- Standard Utilities Programs perform individual specialized management tasks.
  - Shell

- Commands for the management of files and directories
- Filters
- Compilers
- Editors
- Commands for the administration of the system

1.6 The shell of a UNIX System

- The UNIX systems have a Graphical User Interface (Linux uses KDE, GNOME …), but the programmers prefer to type the commands.
- **Shell:** the user process which executes programs (command interpreter)
  - User types command
  - Shell reads command (read from input) and translates it to the operating system.
- Can run external programs (e.g. netscape) or internal shell commands (e.g. cd)
- Various different shells available:
  - Bourne shell (sh), C shell (csh), Korn shell (ksh), TC shell (tcsh), Bourne Again shell (bash).
- The administrator of the system provides to the user a default shell, but the user can change shell.

1.7 Memory Management

Memory management under Linux has two components. The first deals with allocating and freeing physical memory—pages, groups of pages, and small blocks of memory. The second handles virtual memory, which is memory mapped into the address space of running processes.

Management of Physical Memory

Due to specific hardware characteristics, Linux separates physical memory into three different zones identifying different regions of memory. The zones are identified as:

- Z0NE_DMA
- ZONEJTORMAL
- ZONE_HIGHMEM

These zones are architecture specific. For example, on the Intel 80x86 architecture, certain ISA (industry standard architecture) devices can only access the lower 16 MB of physical memory using DMA. On these systems, the first 16 MB of physical memory comprise ZONE-DMA. ZQNEJIORMAL identifies physical memory that is mapped to the CPU's address space. This zone is used for most routine memory requests. For architectures that do not limit what DMA can access, ZONEJDMA is not present, and ZQNEJJQRMAL is used.

Finally, ZONE_HIGHMEM (for "high memory") refers to physical memory that is not mapped into the kernel address space. For example, on the 32-bit Intel architecture (where 232 provides a 4-GB address space), the kernel is mapped into the first 896 MB of the address space; the remaining memory is referred to as high memory and is allocated from ZONE_HIGHMEM.

1.7 File Systems

Linux retains UNIX's standard file-system model. In UNIX, a file does not have to be an object stored on disk or fetched over a network from a remote file server. Rather, UNIX files can be anything capable of handling the input or output of a stream of data. Device drivers can appear as files, and inter process communication channels or network connections also look like files to the user. The Linux kernel handles all these types of file by hiding the implementation details of any single file type behind a layer of software, the virtual file system (VFS). Here, we first cover the virtual file system and then discuss the standard Linux file system—ext2fs.

The Virtual File System

The Linux VFS is designed around object-oriented principles. It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects. The VFS defines four main object types:

- An **inode object** represents an individual file.
- A **file object** represents an open file.
- A **superblock object** represents an entire file system.
- A **dentry object** represents an individual directory entry.

For each of these four object types, the VFS defines a set of operations. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that object. For example, an abbreviated API for some of the file object's operations includes:

- int open (. . .) — Open a file.
- ssize_t read(. . .) —Read from a file.
- ssize_t write (. . .) —Write to a file.
- int mmap (. . .) — Memory-map a file.

The complete definition of the file object is specified in the struct file_operations, which is located in the file /usr/include/linux/fs.h. An implementation of the file object (for a specific file type) is required to implement each function specified in the definition of the file object.

The VFS software layer can perform an operation on one of the file-system objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a networked file, a disk file, a network socket, or a directory file. The appropriate function for that file's readQ operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

The inode and file objects are the mechanisms used to access files. An inode object is a data structure containing pointers to the disk blocks that contain the actual hie contents, and a file object represents a point of access to the data in an open file. A process cannot access an inode's contents without first obtaining a file object pointing to the inode. The file object keeps track of where in the file the process is currently reading or writing, to keep track of sequential file I/O.

It also remembers whether the process asked for write permissions when the file was opened and tracks the process's activity if necessary to perform adaptive read-ahead, fetching file data into memory before the process requests the data, to improve performance. File objects typically belong to a single process, but inode objects do not.

Even when a file is no longer being used by any processes, its inode object may still be cached by the VFS to improve performance if the file is used again in the near future. All cached file data are linked onto a list in the file's inode object. The inode also maintains standard information about each file, such as the owner, size, and time most recently modified.

Directory files are dealt with slightly differently from other files. The UNIX programming interface defines a number of operations on directories, such as creating, deleting, and renaming a file in a directory.

The system calls for these directory operations do not require that the user open the files concerned, unlike the case for reading or writing data. The VFS therefore defines these directory operations in the inode object, rather than in the file object. The superblock object represents a connected set of files that form a self-contained file system. The operating-system kernel maintains a single superblock object for each disk device mounted as a file system and for each networked file system currently connected.

The main responsibility of the superblock object is to provide access to inodes. The VFS identifies every inode by a unique (file-system/inode number) pair, and it finds the inode corresponding to a particular inode number by asking the superblock object to return the inode with that number.

Finally, a dentry object represents a directory entry that may include the name of a directory in the path name of a file (such as /usr) or the actual file (such as stdio.h). For example, the file Aisr/include/stdio. h contains the directory entries (1) /, (2) usr, (3) include, and (4) stdio .h. Each one of these values is represented by a separate dentry object.

As an example of how dentry objects are used, consider the situation in which a process wishes to open the file with the pathname /usr/include/stdio.h using an editor. Because Linux treats directory names as files, translating this path requires first obtaining the inode for the root /. The operating system must then read through this file to obtain the inode for the file include. It must continue this process until it obtains the inode for the file stdio.h. Because path-name translation can be a time-consuming task, Linux maintains a cache of dentry objects, which is consulted during path-name translation. Obtaining the inode from the dentry cache is considerably faster than having to read the on-disk file.

The Linux ext2fs File System

The standard on-disk file system used by Linux is called **ext2fs,** for historical reasons. Linux was originally programmed with a Minix-compatible filesystem, to ease exchanging data with the Minix development system, but that file system was severely restricted by 14-character file-name limits and a maximum file-system size of 64 MB. The Minix file system was superseded by a new file system, which was christened the **extended file system (extfs).** A later redesign of this file system to improve performance and scalability and to add a few missing features led to the **second extended file system (ext2fs).**

2. Windows 2000

Windows 2000 is an operating system for use on both client and servercomputers. It was produced by Microsoft and released to manufacturing on December 15, 1999 and launched to retail on February 17, 2000. It is the successor to Windows NT 4.0, and is the last version of Microsoft Windows to display the "Windows NT" designation.[7] It is succeeded by Windows XP (released in October 2001) and Windows Server 2003 (released in April 2003). During development, Windows 2000 was known as Windows NT 5.0.

Four editions of Windows 2000 were released: Professional, Server,Advanced Server, and Datacenter Server; the latter was both released to manufacturing and launched months after the other editions.

While each edition of Windows 2000 was targeted at a different market, they shared a core set of features, including many system utilities such as the Microsoft Management Console and standard system administration applications.

Support for people with disabilities was improved over Windows NT 4.0 with a number of new assistive technologies, and Microsoft increased support for different languages and locale information.

2.1 The Windows 2000 File System

Windows 2000 supports several file systems, the most important of which are FAT-16, FAT-32, and NTFS. This sample chapter examines the NTFS file system because it is a modern file system unencumbered by the need to be fully compatible with the MS-DOS file system.

Windows 2000 supports several file systems, the most important of which are **FAT-16**, **FAT-32**, and**NTFS** (**NT File System**). FAT-16 is the old MS-DOS file system. It uses 16-bit disk addresses, which limits it to disk partitions no larger than 2 GB. FAT-32 uses 32-bit disk addresses and supports disk partitions up to 2 TB. NTFS is a new file system developed specifically for Windows NT and carried over to Windows 2000. It uses 64-bit disk addresses and can (theoretically) support disk partitions up to 2^{64} bytes, although other considerations limit it to smaller sizes. Windows 2000 also supports read-only file systems for CD-ROMs and DVDs. It is possible (even common) to have the same running system have access to multiple file system types available at the same time.

Fundamental Concepts

Individual file names in NTFS are limited to 255 characters; full paths are limited to 32,767 characters. File names are in Unicode, allowing people in countries not using the Latin alphabet (e.g., Greece, Japan, India, Russia, and Israel) to write file names in their native language. For example, file is a perfectly legal file name. NTFS fully supports case sensitive names (so foo is different from Foo andFOO). Unfortunately, the Win32 API does not fully support case-sensitivity for file names and not at all for directory names, so this advantage is lost to programs restricted to using Win32 (e.g., for Windows 98 compatibility).

An NTFS file is not just a linear sequence of bytes, as FAT-32 and UNIX files are. Instead, a file consists of multiple attributes, each of which is represented by a stream of bytes. Most files have a few short streams, such as the name of the file and its 64-bit object ID, plus one long (unnamed) stream with the data. However, a file can also have two or more (long) data streams as well. Each stream has a name consisting of the file name, a colon, and the stream name, as in foo:stream1. Each stream has its own size and is lockable independently of all the other streams. The idea of multiple streams in a file was borrowed from the Apple Macintosh, in which files have two streams, the data fork and the resource fork. This concept was incorporated into NTFS to allow an NTFS server be able to serve Macintosh clients.

File streams can be used for purposes other than Macintosh compatibility. For example, a photo editing program could use the unnamed stream for the main image and a named stream for a small thumbnail version. This scheme is simpler than the traditional way of putting them in the same file one after another. Another use of streams is in word processing. These programs often make two versions of a document, a temporary one for use during editing and a final one when the user is done. By making the temporary one a named stream and the final one the unnamed stream, both versions automatically share a file name, security information, timestamps, etc. with no extra work.

The maximum stream length is 2^{64} bytes. To get some idea of how big a 2^{64}-byte stream is, imagine that the stream were written out in binary, with each of the 0s and 1s in each byte occupying 1 mm of space. The 2^{67}-mm listing would be 15 light-years long, reaching far beyond the solar system, to Alpha Centauri and back. File pointers are used to keep track of where a process is in each stream, and these are 64 bits wide to handle the maximum length stream, which is about 18.4 exabytes.

The Win32 API function calls for file and directory manipulation are roughly similar to their UNIX counterparts, except most have more parameters and the security model is different. Opening a file returns a handle, which is then used for reading and writing the file. For graphical applications, no file handles are predefined. Standard input, standard output, and standard error have to be acquired explicitly if needed; in console mode they are preopened, however. Win32 also has a number of additional calls not present in UNIX.

3. Windows XP

The Microsoft Windows XP operating system is a 32/64-bit preemptive multitasking operating system for AMD K6/K7, Intel IA32/IA64, and later microprocessors. The successor to Windows NT and Windows 2000, Windows XP is also intended to replace the Windows 95/98 operating system. Key goals for the system are security, reliability, ease of use, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support.

3.1 History

In the mid-1980s, Microsoft and IBM cooperated to develop the OS/2 operating system, which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to make a fresh start and to develop a "new technology" (or NT) portable operating system that supported both the OS/2 and POSIX application-programming interfaces (APIs). In October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building this new operating system. Originally, the team planned for NT to use the OS/2 API as its native environment, but during development, NT was changed to use the 32-bit Windows API (or Win32 API), reflecting the popularity of Windows 3.0.

The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at version 3.1.) Windows NT version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance, with the side effect of decreased system reliability. Although previous versions of NT had been ported to other microprocessor architectures, the Windows 2000 version, released in February 2000, discontinued support for other than Intel (and compatible) processors due to marketplace factors. Windows 2000 incorporated significant changes over Windows NT. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a distributed file system, and support for more processors and more memory.

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In 2002, the server versions of Windows XP became available (called Windows .Net Server). Windows XP updates the graphical user interface (GUI) with a visual design that takes advantage of more recent hardware advances and many new ease-of-use features. Numerous features have been added to automatically repair problems in applications and the operating system itself. Windows XP provides better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video), dramatic performance improvements both for the desktop and large multiprocessors, and better reliability and security than even Windows 2000.

Windows XP uses a client-server architecture (like Mach) to implement multiple operating-system personalities, such as Win32 API and POSIX, with user-level processes called subsystems. The subsystem architecture allows enhancements to be made to one operating-system personality without affecting the application compatibility of any others.

Windows XP is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the graphical user interface via the Windows terminal server. The server versions of Windows XP support simultaneous terminal server sessions from Windows desktop systems. The desktop versions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called fast user switching, allows users to preempt each other at the console of a PC without having to log off and onto the system.

Windows XP is the first version of Windows to ship a 64-bit version. The native NT file system (NTPS) and many of the Win32 APIs have always used 64- bit integers where appropriate—so the major extension to 64-bit in Windows XP is support for large addresses.

There are two desktop versions of Windows XP. Windows XP Professional is the premium desktop system for power users at work and at home. For home users migrating from Windows 95/98, Window's XP Personal provides the reliability and ease of use of Windows XP, but lacks the more advanced features needed to work seamlessly with Active Directory or rim POSIX applications. The members of the Windows .Net Server family use the same core components as the desktop versions but add a range of features needed for uses such as webserver farms, print/file servers, clustered systems, and, large datacenter machines. The large datacenter machines can have up to 64 GB of memory and 32 processors on IA32 systems and 128 GB and 64 processors on IA64 systems.

3.1 Design Principles

Microsoft's design goals for Windows XP include security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support.

Security

Windows XP **security** goals required more than just adherence to the design standards that enabled Windows NT 4.0 to receive a C-2 security classification from the U.S. government (which signifies a moderate level of protection from defective software and malicious attacks). Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities.

Reliability

Windows 2000 was the most reliable, stable operating system Microsoft had ever shipped to that point. Much of this reliability came from maturity in the source code, extensive stress testing of the system, and automatic detection of many serious errors in drivers. The **reliability** requirements for Windows XP were even more stringent.

Microsoft used extensive manual and automatic code review to identify over 63,000 lines in the source files that might contain issues not detected by testing and then set about reviewing each area to verify that the code was indeed correct.

Windows XP extends driver verification to catch more subtle bugs, improves the facilities for catching programming errors in user-level code, and subjects third-party applications, drivers, and devices to a rigorous certification process. Furthermore, Windows XP adds new facilities for monitoring the health of the PC, including downloading fixes for problems before they are encountered by users. The perceived reliability of Windows XP was also improved by making the graphical user interface easier to use through better visual design, simpler menus, and measured improvements in the ease with which users can discover how to perform common tasks.

Windows and POSIX Application Compatibility

Windows XP is not only an update of Windows 2000; it is a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP include a much higher compatibility with consumer applications that run on Windows 95/98. **Application compatibility** is difficult to achieve because each application checks for a particular version of Windows, may have some dependence on the quirks of the implementation of APIs, may have latent application bugs that were masked in the previous system, and so forth. Windows XP introduces a compatibility layer that falls between applications and the Win32 APIs. This layer makes Windows XP look (almost) bug-for-bug compatible with previous versions of Windows.

Windows XP, like earlier NT releases, maintains support for running many 16-bit applications using a thunking, or conversion, layer that translates 16-bit API calls into equivalent 32-bit calls. Similarly, the 64-bit version of Windows XP provides a thunking layer that translates 32-bit API calls into native 64-bit calls. POSIX support in Windows XP is much improved. A new POSIX subsystem called Interix is now available. Most available UNIX-compatible software compiles and runs under Interix without modification.

High Performance

Windows XP is designed to provide **high performance** on desktop systems (which are largely constrained by I/O performance), server systems (where the CPU is often the bottleneck), and large multithreaded and multiprocessor environments (where locking and cache-line management are key to scalability).

High performance has been an increasingly important goal for Windows XP. Windows 2000 with SQL 2000 on Compaq hardware achieved top TPC-C numbers at the time it shipped. To satisfy performance requirements, NT uses a variety of techniques, such as asynchronous I/O, optimized protocols for networks (for example, optimistic locking of distributed data, batching of requests), kernel-based graphics, and sophisticated caching of file-system data.

The memory-management and synchronization algorithms are designed with an awareness of the performance considerations related to cache lines and multiprocessors. Windows XP has further improved performance by reducing the code-path length in critical functions, using better algorithms and per-processor data structures, using memory coloring for NUMA (non-uniform memory access) machines, and implementing more scalable locking protocols, such as queued spinlocks. The new locking protocols help reduce system bus cycles and include lock-free lists and queues, use of atomic read-modify-write operations (like interlocked increment), and other advanced locking techniques.

Extensibility

Extensibility refers to the capacity of an operating system to keep up with advances in computing technology. So that changes over time are facilitated, the developers implemented Windows XP using a layered architecture. The Windows XP executive runs in kernel or protected mode and provides the basic system services. On top of the executive, several server subsystems operate in user mode.

Among them are **environmental subsystems** that emulate different operating systems. Thus, programs written for MS-DOS, Microsoft Windows, and POSIX all run on Windows XP in the appropriate environment.

Because of the modular structure, additional environmental subsystems can be added without affecting the executive. In addition, Windows XP uses loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Windows XP uses a client-server model like the Mach operating system and supports distributed processing by remote procedure calls (RPCs) as defined by the Open Software Foundation.

Portability

An operating system is **portable** if it can be moved from one hardware architecture to another with relatively few changes. Windows XP is designed to be portable. As is true of the UNIX operating system, the majority of the system is written in C and C++. Most processor-dependent code is isolated in a dynamic link library (DLL) called the **hardware-abstraction layer (HAL).** A DLL is a file that is mapped into a process's address space such that any functions in the DLL appear to be part of the process.

3.2 File System

Historically, MS-DOS systems have used the file-allocation table (FAT) file system. The 16-bit FAT file system has several shortcomings, including internal fragmentation, a size limitation of 2 GB, and a lack of access protection for files. The 32-bit FAT file system has solved the size and fragmentation problems, but its performance and features are still weak by comparison with modern file systems. The NTFS file system is much better. It was designed to include many features, including data recovery, security, fault tolerance, large files and file systems, multiple data streams, UNICODE names, sparse files, encryption, journaling, volume shadow copies, and file compression. Windows XP uses NTFS as its basic file system, and we focus on it here. Windows XP continues to use FAT16, however, to read floppies and other removable media. And despite the advantages of NTFS, FAT32 continues to be important for interoperability of media with Windows 95/98 systems. Windows XP supports additional file-system types for the common formats used for CD and DVD media.

POSSIBLE QUESTIONS

UNIT – V

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What do you mean by system calls?
2. What are pipes?
3. What are filters?
4. What is decision making in shell script?.
5. What are utility programs?

PART – C (6 MARKS)

1. Discuss the system calls.
2. Explain the pipes and filters.
3. Explain the decision making in shell script.
4. Discuss the utility programs of Unix and Linux.
5. Explain the pattern matching utility.

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - V : (Objective Type Multiple choice Questions each Question carries one Mark)

OPERATING SYSTEMS

PART - A (Online Examination)

Questions	**Opt1**	**opt2**	**opt3**	**opt4**	**KEY**
computer system assets can be modified only by authorized parities.	Confidentiality	Integrity	Availability	Authenticity	Integrity
In computer security, …………………….. means that the information in a computer system only be accessible for reading by authorized parities.	Confidentiality	Integrity	Availability	Authenticity	Confidentiality
Which of the following is independent malicious program that need not any host program?	Trap doors	Trojan horse	Virus	Worm	Worm
The ……….. is code that recognizes some special sequence of input or is triggered by being run from a certain user ID of by unlikely sequence of events.	Trap doors	Trojan horse	Logic Bomb	Virus	Trap doors
program that is set to “explode” when certain conditions are met.	Trap doors	Trojan horse	Logic Bomb	Virus	Trap doors
Which of the following malicious program do not replicate automatically?	Trojan Horse	Virus	Worm	Zombie	Trojan Horse
…………… programs can be used to accomplish functions indirectly that an unauthorized user could not	Zombie	Worm	Trojan Horses	Logic Bomb	Trojan Horses
programs by modifying them, the modification includes a copy of the virus program, which can go on to infect	Worm	Virus	Zombie	Trap doors	Virus
systems be given just enough privileges to perform their task?	principle of operating system	principle of least privilege	principle of process scheduling	none of the mentioned	principle of least privilege

_______ is an approach to restricting system access to authorized users.	Role-based access control	Process-based access control	Job-based access control	none of the mentioned	Role-based access control
For system protection, a process should access	all the resources	resources for which it has	authorization is not required	all of the mentioned	resources for which it has
The protection domain of a process contains	object name	rights-set	object name and rights-set	none of the mentioned	object name and rights-set
If the set of resources available to the process is fixed throughout the process's lifetime then its domain is	static	dynamic	neither static nor dynamic	none of the mentioned	static
Access matrix model for user authentication contains	a list of objects	a list of domains	a function which returns an object's	all the options	all the options
Global table implementation of matrix table contains	domain	object	right-set	all the options	all the options
For a domain _______ is a list of objects together with the operation allowed on these objects.	capability list	access list	authorization	none of the mentioned	capability list
Which one of the following is capability based protection system?	hydra	cambridge CAP system	hydra & cambridge CAP system	none of the mentioned	hydra & cambridge
In UNIX, domain switch is accomplished via	file system	user	superuser	none of the mentioned	file system
___________ is an important property of an operating system that hopes to keep up with advancements in	Portability	Reliability	Extensibility	compatibility	Extensibility
___________ is the ability to handle error conditions, including the ability of the operating system to protect itself and its users from defective or malicious software	Portability	Reliability	Extensibility	compatibility	Reliability
__________ is the ability to move from one hardware architecture to another with relatively few changes.	Portability	Reliability	Extensibility	compatibility	Portability
Windows NT is designed to afford good ___________.	Portability	Reliability	Extensibility	performance	performance
A __________ is created by the NT disk administrator utility, and is based on a logical disk partition.	Volume	File	Directory	subdirectory	Volume
A __________ of a directory contains the top level of the B+ tree.	index root	file reference	attributes	metadata	index root
The _________ in NT may occupy a portion of a disk, may occupy an entire disk or may span across several	Volume	File	Directory	subdirectory	Volume

The ____________ of a directory contains the top level of the B+ tree.	Volume	File	index root	subdirectory	index root
The ______________ attribute contains the access token of the owner of the file, and an access control list	Portability	Recovery	Reliability	Security	Security
To deal with disk sectors that go bad, ____________ uses a hardware technique called sector spanning.	Ps	Valloc	Kmalloc	FtDisk	FtDisk
places where they have no business being are called __________	intruders	crackers	hackers	worms	intruders
Outsiders can sometimes take command of people's home computers (using viruses and other means) and turn them into ______________	virus	worms	malware	zombies	zombies
who may read and write their files and other objects, This policy is called __________________	mandatory access co	access matrix	discretionary access control.	access control lists	discretionary access control
Every secured computer system must require all users to be ___________________at login time	authenticated	authorized	transferred	scheduled	authenticated
The most widely used form of authentication is to require the user to type a __________and a _______________`	mailid, PIN number	login name, password.	PIN number, Account number	Username, mailid	login name, password.
characteristics of the user that are hard to forge is called as ______________	Biometrics	password	stegnography	access control	Biometrics
____________is the name given to hackers who break into computers for criminal gain	hackers	spoofing	phising	Crackers	Crackers
A typical biometrics system has two parts: ________________	enrollment and identification	identification & authentication	authentication & confidentiality	and authentication	enrollment and identification
Any malware hidden in software or a Web page that people voluntarily download is called _________________	worm	Trojan Horse	Virus	Backdoor	Trojan Horse
The idea of creating a virus that could overwrite the master boot record or the boot sector, with devastating results, such viruses called as ________________	device driver virus	source code virus	companion virus	boot sector viruses	boot sector viruses
The trick to infect a device driver leads to a ________________	source code virus	device driver virus	companion virus	boot sector viruses	device driver virus
When an attempt is to make a machine or network resource unavailable to its intended users, the attack is called	denial-of-service attack	slow read attack	spoofed attack	starvation attack	denial-of-service attack

The code segment that misuses its environment is called a	internal thief	trojan horse	code stacker	none of the mentioned	trojan horse
The internal code of any software that will set of a malicious function when specified conditions are met, is called	logic bomb	trap door	code stacker	none of the mentioned	logic bomb
The pattern that can be used to identify a virus is known as	stealth	virus signature	armoured	multipartite	virus signature
Which one of the following is a process that uses the spawn mechanism to revage the system performance?	worm	trojen	threat	virus	worm
What is a trap door in a program?	a security hole, inserted at programming time	a type of antivirus	security hole in a network	none of the mentioned	a security hole, inserted at programming
Which one of the following is not an attack, but a search for vulnerabilities to attack?	denial of service	port scanning	memory access violation	dumpster diving	port scanning
File virus attaches itself to the	source file	object file	executable file	all of the mentioned	executable file
Multipartite viruses attack on	files	boot sector	memory	all of the mentioned	all of the mentioned
In asymmetric encryption	same key is used for encryption and decryption	different keys are used for encryption and	no key is required for encryption and decryption	none of the mentioned	different keys are used for encryption and
Which of the following are forms of malicious attack ?	Theft of information	Modification of data	Wiping of information	All of the mentioned	All of the mentioned
What are common security threats ?	File Shredding	File sharing and permission	File corrupting	File integrity	File sharing and permission
From the following, which is not a common file permission ?	Write	Execute	Stop	Read	Stop
Which of the following is a good practice ?	permission for remote	only permission	permission to specified account	read and write	permission to specified
What is not a good practice for user administration ?	Isolating a system after a compromise	random auditing procedures	Granting privileges on a per host basis	and FTP for remote access.	Using telnet and FTP for remote access.

Which of the following is least secure method of authentication ?	Key card	fingerprint	retina pattern	Password	Password
Which of the following is a strong password ?	19thAugust88	Delhi88	P@assw0rd	!augustdelhi	P@assw0rd
What does Light Directory Access Protocol (LDAP) doesn’t store ?	Users	Address	Passwords	Security Keys	Address
Which happens first authorization or authentication ?	Authorization	Authentication	Both are same	None of the mentioned	Authorization
What is characteristics of Authorization ?	RADIUS and RSA	handshaking with syn and	protection for securing resources	privileges and rights	privileges and rights
What forces the user to change password at first logon ?	Default behavior of OS	Part of AES encryption practice	Devices being accessed forces the user	Account administrator	Account administrator

Reg No. …………………………….

[16CAU601B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University Established under Section 3 of UGC Act, 1956)

Eachanari Post, Coimbatore - 641 021, India

BCA DEGREE EXAMINATION

(For the candidates admitted from 2016 onwards)

Sixth Semester

First Internal Examination- November 2018

UNIX / LINUX PROGRAMMING

Time: 2 Hours **Maximum: 50 Marks**

Date:

Part - A (20 x 1 =20 Marks)

(Answer all the Questions)

1. Unix was invented in ______________ .
   a. **Bell Laboratories** b. Kell Labs c. Data Labs d. Dell Labs
2. Unix _______ is the core of the operating system.
   .a. Karne b.**Kernel** c.Knot d. Kornel
3. ________ is the distributor of Linux.
   a. White Hat b. Grey Hat c. Data Hat d. **Red Hat**
4. Who invented Linux operating system?.
   a. Lin Lee b.**Linus Torvalds** c. Quin Tom d. Unis Torvalds
5. _________commands are built-in commands in Linux Shell.
   a. External b.File c. Data d. **Internal**

_

6. ________ command is used to check the given command is internal or external.
   a. **type** b. cat c. echo d. print
7. ___________ Commands aren't built into the shell.
   a.Error b. **External** c. cat d. grep
8. Users typically interact with a Unix shell using a ____________.
   a.simulator b. **terminal emulator** c. software emulator d.kernel
9. UNIX supports _________ standard for networking.
   a. UDP b. HTTP c. P2P d. **TCP/IP**
10. Persistent system service processes called _________ in UNIX .
    a.Early b. Exact c. **daemons** d. Error
11. UNIX supports _____________ system.
    a. **virtual file** b. pseudo file c. Fat file d. NTFS File
12. Linux supports GUI using _______ System.
    a.Y Windows b.**X Windows** c. MS Windows d. Lindows
13. Linux is an __________ operating system.

a. open windows b. open Data c. **open source** d. Open command

14. _______ operating system is vulnerable to different types of attacks.

a. UNIX b. LINUX c. Lindows d. **Windows**

15. The advantage of using Linux instead of Windows is _______.

a. interface b. user friendly c. **customization** d. close source

16. _______ Unix command is used to copy files and set file permissions.

a. **Install** b. Rollback c. Transform d. Refresh

17.Users communicate with the kernel through a _______ program.

a.**Shell** b. Kernel c.Shernel d. Vernel

18._______ unix command is used to get information about yourself.

a.user b. **whoami** c. userinfo d. loginuser

19. Linux ______ programs give user most of the functionalities of OS.

a.user b.**utility** c.system d.kernel

20. Linux _______ feature supports users to work on different types of hardware.

a.**portability**b.open source c.multi user d.multi tasking

Part - B (3 x 2=6 Marks)
(Answer all the Questions)

21.What is booting?

Restarting a computer or its operating system software. It is of two types (1) Cold **booting**: when the computer is started after having been switched off. (2) Warm **booting**: when the operating system alone is restarted (without being switched off) after a system crash or 'freeze.'

22.Define kernel in Linux.

The **kernel** is the essential center of a computer operating system (OS). It is the core that provides basic services for all other parts of the OS. It is the main layer between the OS and hardware, and it helps with process and memory management, file systems, device control and networking.

23.Define shell in Unix.\

Shell is a **UNIX** term for the interactive user interface with an operating system.

The **shell** is the layer of programming that understands and executes the commands a user enters. In some systems, the **shell** is called a command interpreter.

Part - C (3 x 8=24 Marks)
(Answer all the Questions)

24.(a) Explain the difference between Unix and Linux.

	Linux	**Unix**
Cost	Linux can be freely distributed, downloaded freely, distributed through magazines, Books etc. There are priced versions for Linux also, but they are normally	Different flavors of Unix have different cost structures according to vendors

	Linux	Unix
	cheaper than Windows.	
Development and Distribution	Linux is developed by Open Source development i.e. through sharing and collaboration of code and features through forums etc and it is distributed by various vendors.	Unix systems are divided into various other flavors, mostly developed by AT&T as well as various commercial vendors and non-profit organizations.
Manufacturer	Linux kernel is developed by the community. Linus Torvalds oversees things.	Three bigest distributions are Solaris (Oracle), AIX (IBM) & HP-UX Hewlett Packard. And Apple Makes OSX, an unix based os..
User	Everyone. From home users to developers and computer enthusiasts alike.	Unix operating systems were developed mainly for mainframes, servers and workstations except OSX, Which is designed for everyone. The Unix environment and the client-server program model were essential elements in the development of the Internet
Usage	Linux can be installed on a wide variety of computer hardware, ranging from mobile phones, tablet computers and video game consoles, to mainframes and supercomputers.	The UNIX operating system is used in internet servers, workstations & PCs. Backbone of the majority of finance infastructure and many 24x365 high availability solutions.
File system support	Ext2, Ext3, Ext4, Jfs, ReiserFS, Xfs, Btrfs, FAT, FAT32, NTFS	jfs, gpfs, hfs, hfs+, ufs, xfs, zfs format
Text mode interface	BASH (Bourne Again SHell) is the Linux default shell. It can support multiple command interpreters.	Originally the Bourne Shell. Now it's compatible with many others including BASH, Korn & C.
What is it?	Linux is an example of Open Source software development and Free Operating System (OS).	Unix is an operating system that is very popular in universities, companies, big enterprises etc.

	Linux	**Unix**
GUI	Linux typically provides two GUIs, KDE and Gnome. But there are millions of alternatives such as LXDE, Xfce, Unity, Mate, twm, ect.	Initially Unix was a command based OS, but later a GUI was created called Common Desktop Environment. Most distributions now ship with Gnome.
Price	Free but support is available for a price.	Some free for development use (Solaris) but support is available for a price.
Security	Linux has had about 60-100 viruses listed till date. None of them actively spreading nowadays.	A rough estimate of UNIX viruses is between 85 -120 viruses reported till date.
Threat detection and solution	In case of Linux, threat detection and solution is very fast, as Linux is mainly community driven and whenever any Linux user posts any kind of threat, several developers start working on it from different parts of the world	Because of the proprietary nature of the original Unix, users have to wait for a while, to get the proper bug fixing patch. But these are not as common.
Processors	Dozens of different kinds.	x86/x64, Sparc, Power, Itanium, PA-RISC, PowerPC and many others.
Examples	Ubuntu, Fedora, Red Hat, Debian, Archlinux, Android etc.	OS X, Solaris, All Linux
Architectures	Originally developed for Intel's x86 hardware, ports available for over two dozen CPU types including ARM	is available on PA-RISC and Itanium machines. Solaris also available for x86/x64 based systems.OSX is PowerPC(10.0-10.5)/x86(10.4)/x64(10.5-10.8)
Inception	Inspired by MINIX (a Unix-like system) and eventually after adding many features of GUI, Drivers etc, Linus Torvalds developed the framework of the OS that became LINUX in 1992. The LINUX kernel was released	In 1969, it was developed by a group of AT&T employees at Bell Labs and Dennis Ritchie. It was written in “C” language and was designed to be a portable, multi-tasking and multi-user system in a time-sharing configuration.

	Linux	**Unix**
	on 17th September, 1991	

(OR)

(b) Discuss the architecture of Unix.

There are two important divisions in UNIX operating system architecture.

1.Kernel

2.Shell

In simple words you can say –

- Kernal – interacts with the machine’s hardware
- Shell – interacts with the user

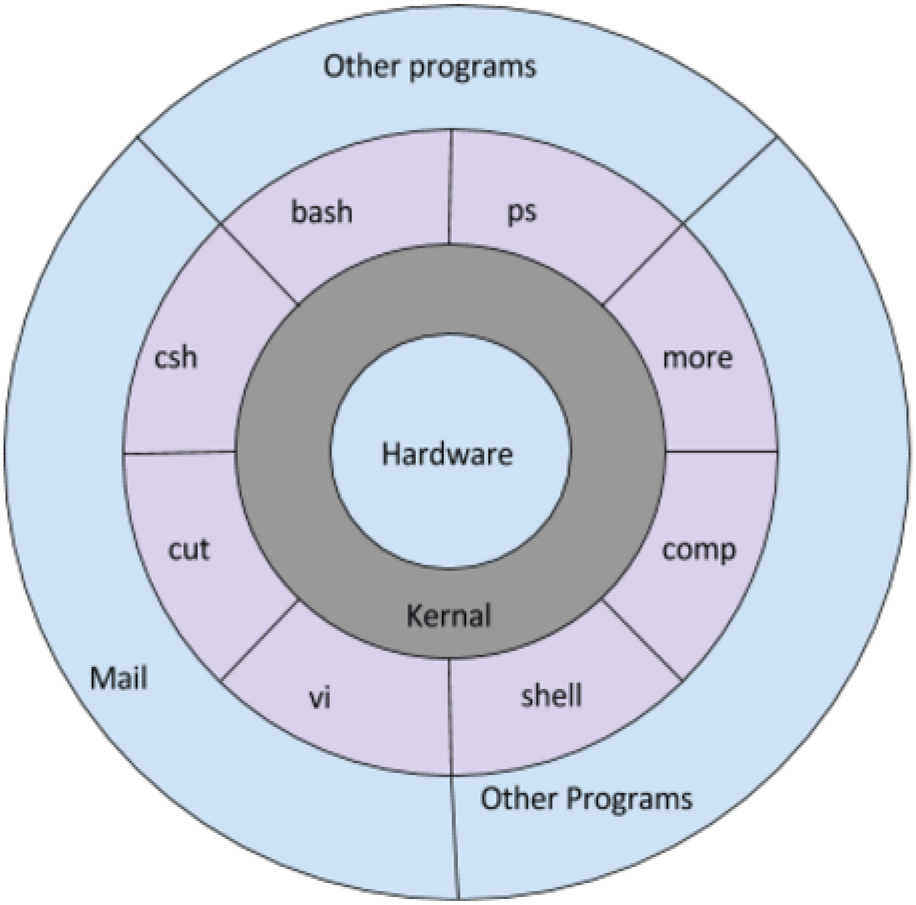
THE KERNEL:

The kernel of UNIX is the hub (or core) of the UNIX operating system. Kernel is a set of routines mostly written in C language.

User programs that need to access the hardware (like hard disk or terminal) use the services of the Kernel, which performs the job on the user’s behalf.

User interacts with the Kernal by using System calls. Kernel allocates memory and time to programs and handles the file store and communications in response to system calls.

As an illustration of the way that the unix shell and the kernel work together, suppose a user types *mv myfile myfile1* (which has the effect of renaming the file myfile). The unix shell searches the file store for the file containing the program mv, and then requests the kernel, through system calls, to execute the program *mv* on *myfile*. When the process *mv myfile* has finished running, the unix shell then returns the UNIX prompt to the user, indicating that it is waiting for further commands.



THE SHELL:
UNIX Shell acts as a medium between the user and the kernel in unix system. When a user logs in, the login program checks the username and password and then starts another program called the shell.
Computers don't have any inherent capability of translating commands into action. This requires a command line interpreter (CLI) and this is handled by the "Outer Part" of the operating system i.e. Shell. It interprets the commands the user types in and arranges for them to be carried out.
In every unix system, the user can customize his own shell, and users can use different shells on the same machine.

25.(a) Explain the architecture of Linux.

Linux System Architecture is consists of following layers:

- **Hardware layer** – Hardware consists of all 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. Takes commands from user and executes kernel's functions.
- **Utilities** – Utility programs giving user most of the functionalities of an operating systems.

(OR)

(b) Discuss the various distributions of Unix and Linux.

Debian

Debian is one of the oldest distributions out there, and some newer, more popular distributions are based on the Debian software. For instance, Ubuntu is a newer Linux distribution based on the Debian architecture. Debian has more than 1,000 volunteers, which makes it very versatile.

Fedora and Red Hat

Red Hat was developed several years ago, and it was known as one of the easiest of the Linux distributions to learn, if you were new to the operating system. Fedora has a great support community, and it's known as one of the better business suites out there.

Ubuntu

For users who are unfamiliar with Linux but want to learn, Ubuntu is the closest to Windows and user-friendly you can get. Ubuntu is one of the babies of the Linux family. It's not that old, but its popularity has grown. Ubuntu has a very "Windows-like" interface, so it's most popular for its ease of use for Windows users who want to migrate to a Linux platform.

Linux Mint

Mint has one of the coolest interfaces for customizing a desktop. Mint's graphics and desktop reminds most old Linux users of the classic GNOME interface. Mint is the alternative to Ubuntu. Most Ubuntu fans who are unhappy with the way Ubuntu is heading moved to Mint.

26.(a) Explain the internal and external commands in Linux.

The UNIX/LINUX system is command-based *i.e* things happen because of the commands that you key in. All UNIX commands are seldom more than four characters long.

They are grouped into two categories: 1.Internal commands 2.External Commands

- **Internal Commands :** Commands which are built into the shell. For all the shell built-in commands, execution of the same is fast in the sense that the shell doesn't have to search the given path for them in the PATH variable and also no process needs to be spawned for executing it.
  Examples: source, cd, fg etc.

INTERNAL COMMANDS IN UNIX/LINUX
1.Date command is used to display the date
Ex: $date – date="yesterday"

2.Echo command is used to display any message

$echo "Connect your pc in network"

3. printf ("print formatted") command is a shell builtin (and utility program) that formats and prints data. It accepts a format string, which specifies methods for formatting items, and a list of items to be formatted. Named for the intention of printing to a printer, it actually outputs to stdout.

4.type command is used to check whether the command is external or internal.

$type cd
Cd is shell builtin

5.cd command is used to change the directory
$cd prog

6. **stty** displays or changes the characteristics of the terminal.
$stty

7.Who am I is command used to display the current user.

8.man command is used to help documents of the specific command.

$man type

External Commands in UNIX/LINUX : Commands which aren't built into the shell. When an external command has to be executed, the shell looks for its path given in PATH variable and also a new process has to be spawned and the command gets executed.

(OR)

(b) Explain the Installation, Booting and shutdown process of Unix.

Installation

The default installation process is text-based, but you can use an option that activates a graphical user interface during installation.

The following steps are common to all UNIX operating systems. Once you perform them, you are directed to the procedure for your particular operating system to complete the installation.

When installing using the server installation CD you are asked to provide the location of your license.xml file. If you have purchased a license from BEA, provide the complete path to the license.xml file when prompted. If you are evaluating the software, leave the path blank. A a 30-day evaluation license will be created for you

The boot process

When an x86 computer is booted, the processor looks at the end of the system memory for the BIOS (Basic Input/Output System) and runs it. The BIOS program is written into permanent read-only memory and is always available for use. The BIOS provides the lowest level interface to peripheral devices and controls the first step of the boot process.

Shutdown

UNIX was not made to be shut down, but if you really must, use the **shutdown** command. After completing the shutdown procedure, the -h option will halt the system, while -r will reboot it.

The **reboot** and **halt** commands are now able to invoke **shutdown** if run when the system is in run levels 1-5, and thus ensure proper shutdown of the system,but it is a bad habit to get into, as not all UNIX/Linux versions have this feature.

Reg No. …………………………….

[16CAU601B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University Established under Section 3 of UGC Act, 1956)

Eachanari Post, Coimbatore - 641 021, India

BCA DEGREE EXAMINATION

(For the candidates admitted from 2016 onwards)

Sixth Semester

Second Internal Examination- February 2019

UNIX / LINUX PROGRAMMING

Time: 2 Hours **Maximum: 50 Marks**

Date:

Part - A (20 x 1 =20 Marks)

(Answer all the Questions)

1. _______ acts like an interface between user and computer.
   a. **Operating System** b. Device Driver c. file software d. Hardware
2. When we press restart button, ______ will occur.
   a.Water boot b.**warm booting** c.cold booting d. boil booting
3. ________ is the distributor of Linux.
   a. White Hat b. Grey Hat c. Data Hat d. Fedora
4. Unix was written originally in _____ language .
   a. Fortran b.**assembly** c.Java d. C++
5. _________commands are not built-in commands in Unix Shell.
   a. External b.File c. Data d. **external**
6. ________ is an example for Unix system utility.
   a. **mkfs** b. rkfs c. ekfs d. pkfs
7. __________ Command is used to create directories.
   a. man b. **mkdir** c. cat d. cd
8. When we press power button, _______ will occur.
   a.Water boot b.**warm booting** c.cold booting d. boil booting
9. Unix based _______ system is more efficient than DOS based computers.
   a. hardware b. software c. network d. **multi processor**
10. Unix was invented in _______ .
    a. Axe labs b. dell labs c. **bell labs** d.hell labs
11. In UNIX, _______ command is used to suspend execution for an interval.
    a. **sleep** b. delay c. wait d. kill
12. The primary job of Unix OS is to ______.
    a. Manage user b. **manage resource** c. give utility d.manage memory

13. The special purpose OS in Linux is called as _______.
a.fake time OS b. Secure OS c. **real time OS** d.powerful OS

14.Which of the following Operating System has less number of virus?
a. UNIX b. **LINUX** c. Lindows d. Windows

15 ._______ file is used for mounting disk partitions.
a. /dtc/fstab b. /etc/fstac c. **/etc/fstab** d. /etc/fstb

16.Linux OS implements _____ mechanism to create a new process.
a. **Spawn** b. spin c. zombie d. spavn

17.In Unix, the ____ of child process is overwritten with the new process data.
a.memory b. **address space** c.daemon d. Vernel

18._______ unix command is used to set the file system on hard disk.
a.**mkfs** b. ckfs c. skfs d. dkfs

19. Linux ______ programs give user most of the functionalities of OS.
a.user b.**utility** c.system d.kernel

20. The init process will start after the ______ process of Linux booting.
a.file b.**booting** c.memory d.checking

Part - B (3 x 2=6 Marks)
(Answer all the Questions)

21.What is fork command in Unix?
22.What are the types of users in Linux?
23.Define zombie process in Unix.

Part - C (3 x 8=24 Marks)
(Answer all the Questions)

24.(a) Explain the role of fork, exec, and wait function in Linux process creation.
(OR)
(b) Explain the various types of process in Unix.

.

25.(a) Explain user management commands.
(OR)
(b) Discuss File Quota and various file systems available in Unix.

26.(a) Explain file system management in Unix.
(OR)
(b) Explain the role of hard links and symbolic links in Linux.

Reg No. …………………………….

[16CAU601B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University Established under Section 3 of UGC Act, 1956)

Eachanari Post, Coimbatore - 641 021, India

BCA DEGREE EXAMINATION

(For the candidates admitted from 2016 onwards)

Sixth Semester

Second Internal Examination- February 2019

UNIX / LINUX PROGRAMMING

Time: 2 Hours **Maximum: 50 Marks**

Date:

Part - A (20 x 1 =20 Marks)
(Answer all the Questions)

1. _______ acts like an interface between user and computer.
   a. **Operating System** b. Device Driver c. file software d. Hardware
2. When we press restart button, ______ will occur.
   a.Water boot b.**warm booting** c.cold booting d. boil booting
3. ________ is the distributor of Linux.
   a. White Hat b. Grey Hat c. Data Hat d. Fedora
4. Unix was written originally in _____ language .
   a. Fortran b.**assembly** c.Java d. C++
5. _________commands are not built-in commands in Unix Shell.
   a. External b.File c. Data d. **external**
6. ________ is an example for Unix system utility.
   a. **mkfs** b. rkfs c. ekfs d. pkfs
7. __________ Command is used to create directories.
   a. man b. **mkdir** c. cat d. cd
8. When we press power button, _______ will occur.
   a.Water boot b.**warm booting** c.cold booting d. boil booting
9. Unix based _______ system is more efficient than DOS based computers.
   a. hardware b. software c. network d. **multi processor**
10. Unix was invented in _______ .
    a. Axe labs b. dell labs c. **bell labs** d.hell labs
11. In UNIX, _______ command is used to suspend execution for an interval.
    a. **sleep** b. delay c. wait d. kill
12. The primary job of Unix OS is to ______.
    a. Manage user b. **manage resource** c. give utility d.manage memory

13. The special purpose OS in Linux is called as _______.

a.fake time OS b. Secure OS c. **real time OS** d.powerful OS

14.Which of the following Operating System has less number of virus?

a. UNIX b. **LINUX** c. Lindows d. Windows

15 ._______ file is used for mounting disk partitions.

a. /dtc/fstab b. /etc/fstac c. **/etc/fstab** d. /etc/fstb

16.Linux OS implements _____ mechanism to create a new process.

a. **Spawn** b. spin c. zombie d. spavn

17.In Unix, the ____ of child process is overwritten with the new process data.

a.memory b. **address space** c.daemon d. Vernel

18._______ unix command is used to set the file system on hard disk.

a.**mkfs** b. ckfs c. skfs d. dkfs

19. Linux ______ programs give user most of the functionalities of OS.

a.user b.**utility** c.system d.kernel

20. The init process will start after the ______ process of Linux booting.

a.file b.**booting** c.memory d.checking

Part - B (3 x 2=6 Marks)

(Answer all the Questions)

21.What is fork command in Unix?

22.What are the types of users in Linux?

23.Define zombie process in Unix.

Part - C (3 x 8=24 Marks)

(Answer all the Questions)

24.(a) Explain the role of fork, exec, and wait function in Linux process creation.

(OR)

(b) Explain the various types of process in Unix.

.

25.(a) Explain user management commands.

(OR)

(b) Discuss File Quota and various file systems available in Unix.

26.(a) Explain file system management in Unix.

(OR)

(b) Explain the role of hard links and symbolic links in Linux.

Reg No. …………………………….
[16CAU601B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University Established under Section 3 of UGC Act, 1956)

Eachanari Post, Coimbatore - 641 021, India

BCA DEGREE EXAMINATION
(For the candidates admitted in 2016 onwards)
Sixth Semester
Third Internal Examination – March 2019
UNIX / LINUX PROGRAMMING

Duration : 2 Hours | **Maximum: 50 Marks**
Date &Session : 11.3.2019 & FN | **Class: III BCA**

Part - A (20 x 1 =20 Marks)
(Answer all the Questions)

1. In Linux, _______ is used to prevent process for using critical resource.
   a. Semaphore b. preemptive c**. file management** d. Disk process
2. In Unix, ______ resource is shared by many processes.
   a.semaphore b.**critical section** c.x window d. Kernel
3. ________ is the distributor of Linux.
   a. White Hat b. **Grey Hat** c. Data Hat d. suse
4. $date-date will display _______ .
   a. today date b.**yesterday date** c.tomorrow date d.error
5. _________commands are not built-in commands in Unix Shell.
   a. External b.File c. Data d. **external**
6. ________ command is used to join files.
   a. **cat** b. rat c. est d. pst
7. __________ Command is used to create directories.
   a.man b. **mkdir** c. cat d. cd
8. _____ command is used to set the environment for command invocation.
   a.snv b**. env** c. inv d.knv
9. In Linux, _______ command is used to terminate or signal processes.
   a. del b. destroy c. delete d. **kill**
10. In UNIX, _________ command is used to remove directory entries .
    a.nm b. fm c. **rm** d.gm
11. In UNIX, _______ command is used to suspend execution for an interval.
    a**. sleep** b. delay c. wait d. kill
12. Unix uses two distinct functions __________ for creating processes.
    a.pork() and qxec() b. **fork() and exec()** c. del() and kill() d.lork() and cark()

13. Windows is more __________ operating system comparing to Linux OS.
a.faster b. secure c. **user friendly** d.powerful

14.Which of the following Operating System has less number of virus?
a. UNIX b. **LINUX** c. Lindows d. Windows

15 ._______ file is used for mounting disk partitions.
a. /dtc/fstab b. /etc/fstac c. **/etc/fstab** d. /etc/fstb

16.Linux OS implements _____ mechanism to create a new process.
a. **Spawn** b. spin c. zombie d.spavn

17.In Unix, the ____ of child process is overwritten with the new process data.
a.memory b. address space c.**daemon** d. Vernel

18._______ unix command is used to set the file system on hard disk.
a.**mkfs** b. ckfs c. skfs d. dkfs

19. Linux ______ programs give user most of the functionalities of OS.
a.user **b.utility** c.system d.kernel

20. The init process will start after the ______ process of Linux booting.
a.file **b.booting** c.memory d.checking

Part - B (3 x 2=6 Marks)
(Answer all the Questions)

21.What is hard link in Unix?

A **hard link** is merely an additional name for an existing file on **Linux** or other Unix-like operating systems. Any number of **hard links**, and thus any number of names, can be created for any file.

22. What are the benefits of shell scripts?

- Creating your own power tools/utilities.
- Automating command input or entry.
- Customizing administrative tasks.
- Creating simple applications.

23.What is pattern matching utility in Linux?

Pattern matching utility is used to search for a specific terms. Pattern matching commands are **grep** and **egrep**. Pattern match utility is used to search *regular expressions* contain normal characters mixed with special characters

Part - C (3 x 8=24 Marks)
(Answer all the Questions)

24. (a) Discuss the role of pipes and filters in Unix.

Pipes and Filters

The purpose of this lesson is to introduce you to the way that you can construct powerful Unix command lines by combining Unix commands.

Concepts

Unix commands alone are powerful, but when you combine them together, you can accomplish complex tasks with ease. The way you combine Unix commands is through using pipes and filters.

Using a Pipe

The symbol | is the Unix pipe symbol that is used on the command line. What it means is that the standard output of the command to the left of the pipe gets sent as standard input of the command to the right of the pipe. Note that this functions a lot like the > symbol used to redirect the standard output of a command to a *file*. However, the pipe is different because it is used to pass the output of a command to *another command*, not a file.

Here is an example:

```
$ cat apple.txt
core
worm seed
jewel
$ cat apple.txt | wc
      3       4      21
$
```

In this example, at the first shell prompt, I show the contents of the file apple.txt to you. In the next shell prompt, I use the cat command to display the contents of the apple.txt file, but I sent the display not to the screen, but through a pipe to the wc (word count) command. The wc command then does its job and counts the lines, words, and characters of what it got as input.

You can combine many commands with pipes on a single command line. Here's an example where I count the characters, words, and lines of the apple.txt file, then mail the results to nobody@december.com with the subject line "The count."

```
$ cat apple.txt | wc | mail -s "The count" nobody@december.com
```

Using a Filter

A filter is a Unix command that does some manipulation of the text of a file. Two of the most powerful and popular Unix filters are the sed and awk commands. Both of these commands are extremely powerful and complex.

sed

Here is a simple way to use the sed command to manipulate the contents of the apple.txt file:

(OR)

(b).Explain the two types of shell varaibles..

User Defined Variables in Linux

These variables are defined by **users**. A shell script allows us to set and use our **own variables** within the script. Setting variables allows you to **temporarily store data** and use it throughout the script, making the shell script more like a real computer program. **User variables** can be any text string of up to **20 letters**, **digits**, or **an underscore character**. User variables are case sensitive, so the variable Var1 is different from the variable var1. This little rule often gets novice script programmers in trouble.
Values are assigned to user variables using an **equal sign.** No spaces can appear between the variable, the equal sign, and the value (another trouble spot for novices). Here are a few examples of assigning values to user variables:

```
var1=10
var2=-57
var3=testing
var4="still more testing"
```

The shell script **automatically determines the data type** used for the variable value. Variables defined within the shell script maintain their values throughout the life of the shell script but are deleted when the shell script completes.

System Variables in Linux

Environment variables are dynamic values which affect the processes or programs on a computer. They exist in every operating system, but types may vary. Environment variables can be created, edited, saved, and deleted and give information about the system behavior.

Environment variables can change the way a software/programs behave.

E.g. $LANG environment variable stores the value of the language that the user understands. This value is read by an application such that a Chinese user is shown a Mandarin interface while an American user is shown an English interface.

25.(a)Discuss the bourne, bash, csh. tcsh shells in Unix

Bash Shell

Bash is a Unix shell. It was created as a substitute for Bourne shell and include much more scripting tools than Bourne shell like the csh and ksh shells.

Bash is a very common shell and you actually might be running it by default on your machine. It is almost always available on all Linux distributions. One of the contender to Bash shell is dash which is becoming more popular by the Ubuntu project.

Csh Shell

Bill Joy created it at the University of California at Berkeley. It incorporated features such as aliases and command history. It includes helpful programming features like built-in arithmetic and C-like expression syntax.

Csh is an improved C shell. It is most popular in terms of a login shell and shell command interpreter. Most favorable features of this shell are:

- Syntax similar to C
- Control over jobs
- Intelligent spell correction
- Command-line editor
- Filename completion

The Bourne Shell –

Denoted as **sh**

It was written by Steve Bourne at AT&T Bell Labs. It is the original UNIX shell. It is faster and more preferred. It lacks features for interactive use like the ability to recall previous commands. It also lacks built-in arithmetic and logical expression handling. It is default shell for Solaris OS.

For the Bourne shell the:

Command full-path name is /bin/sh and /sbin/sh,

Non-root user default prompt is $,

Root user default prompt is #.

(OR)

(b).Discuss the command mode, insert mode and line mode of Unix Vi Editor.

- **Command Mode:** When vi starts up, it is in Command Mode. This mode is where vi interprets any characters we type as commands and thus does not display them in the xterm window. This mode allows us to move through a file, and to delete, copy, or paste a piece of text. To enter into Command Mode from any other mode, it suffices to press the [Esc] key. If we press [Esc] when we are already in Command Mode, then vi will beep or flash the screen.

- **Input Mode**: In Input Mode, vi accepts keystrokes as text and displays the text as it is entered from the keyboard. vi must be in Input Mode before we can insert text into a file. To enter into Input Mode, we need to put vi into Command Mode and type the key [i].
- **Line Mode**: Line Mode is invoked by typing a colon [:] or a slash [/] while vi is in Command Mode. The cursor will jump to the last line of the screen and vi will wait for a command.

26.(a) Explain if else and switch decision making shell scripts in Linux.

Shell Script using if else if

```
#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

   echo "a is equal to b"

else

   echo "a is not equal to b"

fi
```

```
SHELL SCRIPT USING SWITCH CASE
#!/bin/sh

echo "Please talk to me ..."
while:
do
  read INPUT_STRING
case $INPUT_STRING in
                hello)
                                echo "Hello yourself!"
                                ;;
                bye)
                                echo "See you again!"
                                break
                                ;;
                *)
                                echo "Sorry, I don't understand"
```

```
                    ;;
esac
done
echo
echo "That's all folks!"
```

(OR)

(b).Explain For Loop and while loop shell scripting.

```
#!/bin/sh


a=0

while["$a"-lt10]# this is loop1

do

  b="$a"

while["$b"-ge0]# this is loop2

do

    echo -n "$b "

    b=`expr $b - 1`

done

  echo

  a=`expr $a + 1`

done

#!/bin/sh


for var1 in123

do

for var2 in05

do

if[ $var1 -eq2-a $var2 -eq0]
```

```
then

break2

else

        echo "$var1 $var2"

fi

done

done
```