Chapter : 1

Introduction to Operating Systems

1.0 OBJECTIVES

After studying this chapter you will be able to -

- Discuss feature of Operating Systems.
- To Study and understand Desktop Operating System and Network Operting System.
- Discuss Ms Dos and Windows Operating System.

1.1 INTRODUCTION

Operating System Basics

An Operating system is a part of system software that is loaded in to computer on boot up that is responsible for running other applications and provides interface to interact with other programs that uses system hardware. This interface is either command line user interface or Graphical User Interface. Command Line User interface is used in operating systems like MSDOS, UNIX, LINUX etc. and GUI is used with most of MS Windows operating systems like Windows XP , Windows Vista Windows 7 etc.

1.2 FEATURES

Basic Features

1] File Management
2] Print Management
3] Process Management
4] I/O Management
5] Disk Management

Operating system can be divided into two groups :

1] Single process & 2] Multi process.

Single process operating systems are capable of working on one task at a time while multi process operating systems can work on several processes at once by breaking the tasks into threads. Smallest part of programs that can be scheduled for execution is called as a thread. There are several terms related to multiprocessing which are as follows

Multitasking:- It is the capability of an operating system to handle more than one task at a time. There are two types of multitasking

1] Co-operative Multitasking

2] Preemptive multitasking.

1) Co-operative Multitasking - Applications can control the system resource until they are finished. If a task caused faults or other problems, it would cause the system to become unstable and force a reboot. This type of multitasking is used in Windows 3.x.

2) Pre-emptive Multitasking - Applications are allowed to run for a specified period of time depending on how important the application is to the operation of the system (priority basis). If a particular task is causing problems or faults, that application can be stopped without the system becoming unstable. Used in Windows 9.x, Windows XP, Vista and Windows 7 and all network operating systems.

Multi user - This is similar to multitasking and is the ability for multiple users to access resources at the same time. The OS switches back and forth between users. For example all network operating systems like Windows server 2003, Windows server 2008, Linux, Unix etc.

Multiprocessor - Having multiple processors installed in a system such that tasks are divided between them. Now all latest operating systems uses symmetric multiprocessing.

Multiprogramming - It is the capability of an operating system to run multiple programs at once. Multiprogramming is possible because of multi threading.

Multithreading - It is the capability of an operating system to handle (execute) multiple threads of multiple programs at a time. One program may have at least one thread. Thread is a smallest part of a program that can be scheduled for execution.

There are two broad categories of operating systems 1] Desktop operating system 2] Network operating system.

1.1 & **1.2** **Check Your Progress.**

I State Whether the following statements are true or false

1] Operating system is a part of Application software.

2] Smallest part of programs that can be scheduled for execution is called as a thread.

3] All operating systems support Multi User feature.

4] Multiprocessing supports multiple processors.

1.3 DESKTOP OPERATING SYSTEM

Features of Desktop operating system

1] It a single user operating system.

2] It can support Multitasking, Multiprogramming, Multiprocessing and Multithreading. 3] User interface can be command line or GUI.

4] Desktop PCs, workstations and laptops are used to install desktop operating systems.

5] Cost of the operating system is low as compare to Network operating system.

6] Desktop operating system can be configured as a client in network environment.

7] Provides user level and share level security.

8] Desktop operating systems are as follows :
MSDOS, Windows 95/98, Windows 2000 Professional, Windows XP, Windows Vista and Windows 7.

1.4 NETWORK OPERATING SYSTEM

Features of NOS

1] It is a multi user operating system.

2] It is also called as server operating system.

3] It can support Multitasking, Multiprogramming, Multiprocessing and Multithreading and Multi User.

4] User interface can be Command Line or GUI.

5] Generally server grade computers are used for installation of network operating system.

6] Cost of network operating system is very high as compared to desktop operating system.

7] It provides high security that includes user level, share level as well as file level.

8] It provides backup tools to take the backup of important data. Backup can be scheduled in non working hours also.

9] It can be configured as a server to provide various types of network services like file, print, database, mail, proxy, web etc.

10] Network operating systems are as follows:

Windows Server 2000, Windows Server 2003, Windows Server 2008. Red Hat Enterprise Linux Server 4.0, and 5.0, UNIX, Novell's Netware 4.x, 5.x and 6.x servers etc.

Apart from being as a User interface between system hardware and User applications operating system have other functions which are as follows

1] Memory management 2] Process Management 3] Disk Management 4] I/O management 5] Device Management 6] Backup Management 7] Security Management.

1.3 & **1.4** **Check Your Progress.**

Fill in the Blanks

1] Desktop operating System is a ----------------- user operating system.

2] Desktop operating system can be configured as a ----------- in network environment.

3] Network operating system is a ---------- user operating system.

4] Cost of network operating systems are -------- as compared to Desktop operating system.

1.5 MS-DOS Structure

Disk Operating System (DOS) is a single user single-process operating system that uses a command line interface known as a DOS prompt. Files with .COM, .BAT and .EXE can be executed from the prompt.

The following is a list of DOS system files in the order that they are called during the bootstrap process:

1] IO.SYS - Located in the Root directory and defines basic Input/Output routines for the processor. It is a Hidden and Read Only file. This is required for OS start-up. IO.SYS runs MSDOS.SYS, CONFIG.SYS and then COMMAND.COM.

2] MSDOS.SYS - Located in the Root and defines system file locations. It is Hidden and Read Only. This is required for OS start-up.

3] CONFIG.SYS - Located in the Root and automatically loaded by MSDOS.SYS. This loads low level device drivers for hardware and memory drivers such as HIMEM.SYS and EMM386.EXE. Other drivers include ANSI.SYS, DISPLAY.SYS, KEYBOARD.SYS, PRINTER.SYS and DRIVER.SYS which assigns drive letters to floppy drives. CONFIG.SYS is not required for OS Start-up.

4] **HIMEM.SYS -** Controls Extended Memory management in the extended memory area. Located in C:\DOS and is not required for OS start-up.

5] **EMM386.EXE -** Controls Expanded memory in the upper memory area. Located in C:\DOS and is not required for OS start-up.

6] **COMMAND.COM -** This is the command Interpreter. It is responsible for the command prompt and contains all the internal commands such as DIR,COPY, and CLS. Located normally in the Root directory but can be located elsewhere and specified in the Autoexec.bat with a "SET COMSPEC=". This carries no attributes and is required for OS start-up.

7] **AUTOEXEC.BAT -** Located in the Root and automatically executed at startup.

Runs Programs (Prompt, WIN, CLS etc) and set commands (Path, Comspec etc..). Also calls other batch files. This is not required for OS Start-up.

The DOS interface is a command line prompt at which commands are entered and can utilize wildcards such as the asterisk(*). Many of the DOS commands are internal which means that they are stored in COMMAND.COM. The external commands are supplemental utilities.

Memory Management of DOS

- First 640k is Conventional Memory
- 640k to 1024k is Upper Memory
- Above 1024k is Extended Memory
- HIMEM.SYS is loaded in CONFIG.SYS as the first driver to manage the Extended Memory are and to convert this to XMS (Extended Memory Specification). The first 64k of extended memory has been labelled High Memory (HMA). DOS can be put here by putting DOS=HIGH in CONFIG.SYS.
- EMM386.EXE is loaded in CONFIG.SYS after HIMEM.SYS has been successfully loaded. This is used in the hardware reserved 384k of space in upper memory (640k-1024k) and creates EMS(Extended Memory Specification).
- Virtual Memory relies upon EMS (thereforeEMM386.EXE) and uses hard disk space as memory. SMARTDRV.SYS is a disk caching program for DOS and Windows 3.x systems. The smartdrive program keeps a copy of recently accessed hard disk data in memory. When a program or MSDOS reads data, smartdrive first checks to see if it already has a copyand if so supplies it instead of reading from the hard disk.

1.6 MICROSOFT WINDOWS OPERATING SYSTEMS

Windows 9x Structure

Windows 95 and 98 are 32-bit operating systems. Windows 95 had 2 releases - The FAT16 original release and later OSR2 which utilized the FAT32 file system, added personal web server, Active Desktop and several other new features.

Windows 98 had 2 releases as well - The original version and Windows 98 Second Edition(SE).

Below is an outline of the boot process and the files involved.

1. POST - Hardware tests
2. Plug and Play Configuration -Windows 9x is a Plug and Play(PnP) operating system. In order for PnP to work, the BIOS, hardware and operating system must all be PnP compliant.
3. Master Boot Record - The MBR is located.
4. IO.SYS - This file loads drivers and executes CONFIG.SYS, MSDOS.SYS and COMMAND.COM
5. COMMAND.COM - Loads AUTOEXEC.BAT
6. Windows core files are loaded

   _ WIN.COM - This file begins the loading of Windows 9x system files.

   _ KERNEL32.DLL/KERNEL386.EXE - These files contain the core operating system and is responsible for loading device drivers.

   _ GDI.EXE/GDI32.EXE - These files are responsible for loading the basic GUI or graphical user interface.

   _ WIN.INI - Along with WINFILE.INI and SYSTEM.INI, these files provide backward compatibility with older 16-bit applications and are not required in order for 32- bit applications to run under Windows 9x. Most of the functions of these files are now stored in the registry files.
7. The startup folder is checked for applications to load on startup. Windows 9x also replaces many of the DOS start-up files such as IO.SYS, MSDOS.SYS and COMMAND.COM with newer versions. Most of the functions of the CONFIG.SYS and AUTOEXEC.BAT files are now handled by the new IO.SYS file, although entries in CONFIG.SYS will take precedence over entries in the IO.SYS file. Windows 9x supports long file names up to 255 characters. Duplicate filenames (8.3 format) are assigned for backward compatibility (i.e. DOS). This is done by taking the 1st 6 characters of the filename adding a tilde and then a number. For example,

   VERYLONGFILENAME.EXE would become

   VERYLO~1.EXE.

Windows 9x Core Components

The Windows 9x family of operating systems are made up of 3 core components:

- Kernel - OS foundation that provides error handling, virtual memory management, task scheduling, and I/O services.

- User - Responsible for the user interface and handles input from hardware devices by interacting with device drivers.
- GDI - Responsible for what appears on the display and other graphics functions.

Windows 9x Registry

The registry is a hierarchical database that contains the system's configuration information. The registry is made up of 2 files that are located in the Windows directory:

- USER.DAT - Contains information about system users.
- SYSTEM.DAT - Contains information about hardware and system settings.

The registry can be accessed with the utility REGEDIT.EXE which allows you to edit and restore the registry settings. The Windows 95 registry is backed up every time the machine is booted. The backup files are called USER.DA0 and SYSTEM.DA0. Windows 98 backs up SYSTEM.DAT, USER.DAT, SYSTEM.INI and WIN.INI as .CAB files that are stored in the hidden Windows\Sysbackup directory.

Windows NT/2000 Structure

Windows NT and 2000 are 32 bit operating systems that run in 2 different modes which are kernel(protected) and user. Applications use Application Program Interfaces(APIs) to pass threads between the 2 modes. User mode provides no direct access to the system's hardware. The subsystems of these operating systems are outlined below.

- WIN32 -- This subsystem handles support for 32-bit windows applications and is also known as the Client/Server subsystem. This subsystem has the following features:
  - 32-bit architecture
  - Multiple execution threads are supported for each process
  - Memory Protection - each Win32 application is separated and protected from other applications
  - OpenGL - Support for 2D and 3D graphics cards
  - 2GB non segmented address spaces are assigned to each application
  - NT/2000 supports DOS applications via VDMs(Virtual DOS Machines).

A VDM is a Win32 application that creates an environment where DOS applications can run. It does this by making the NT Workstation resemble a DOS environment and tricks the DOS applications into thinking that they have unrestricted access to the computer's hardware. NT can only support DOS applications that use VDDs (Virtual Device Drivers) to intercept the applications calls to the computer's hardware.

  - NT/2000 also supports Win16 applications with the use of a DOS application called WOW(Windows on Windows). WOW runs within a VDM that runs as a 32-bit process. If a Win16 application crashes it will only corrupt the WOW, but will not affect the rest of the NT operating system. In addition to the above, Windows 2000 also adds the Plug and Play Manager and the Power Management Manager. The boot files used by NT/2000 are completely different than Windows 9x and are listed below:

- BOOT.INI - Specifies boot defaults, operating system locations, settings and menu selections.
- BOOTSECT.DOS - A file located in the system partition that allows the option to boot into another operating system such as Win98 or DOS.
- NTDETECT.COM - Hardware detection program that is located on the root of the system partition.
- NTLDR - File that loads the operating system and is located on the root of the system partition.
- NTOSKRNL.EXE - The executable file.
- OSLOADER.EXE - This is the OS loader for RISC based systems.
- NTBOOTDD.SYS - File used when the system or boot partition is located on a SCSI drive and the BIOS is disabled.

The registry editors included with Windows NT/2000 include Regedt32 and Regedit. For Windows 2000, the Regedt32 tool should be used while Windows NT can use either. Most of the registry(the static items) are contained in hive files which are located in the \WINNT\SYSTEM32\CONFIG directory. The 5 hives are SAM, security, software, system and default.

In Windows 2000 most system administration tasks are performed in the Computer Management Console that contains all of the various Microsoft Management Consoles(MMCs) in one location.

Windows 2000 filenames can be up to 215 characters long and cannot contain the following: <>\/?*"| Windows 2000 supports PnP while NT does not.

Windows XP Structure

Windows XP operating system is either 32 bit or 64 bit operating system that run in 2 different modes which are kernel(protected) and user. Applications use Application Program Interfaces(APIs) to pass threads between the 2 modes. User mode provides no direct access to the system's hardware.

Windows XP has various editions like Windows XP Home, Windows XP Professional, Windows XP Media Centre etc.

Windows XP support 2 way SMP i.e. two multiprocessors.

Windows XP Home edition can not take part in domain networks.

Features of Windows XP :-

Windows XP combines the features of Windows 2000 Professional, Windows 98 and Windows ME. The main benefit of using XP include its reliability, performance, security, ease of use, support for remote users, networking and communication support, management and deployment capabilities, and help and support features.

Hardware Requirements

To install Windows XP Professional successfully, your system must meet certain hardware requirements. The standard Windows XP Professional operating system is based on the Intel x86-based processor architecture, which uses a 32-bit operating system.

Windows XP 64-bit edition is the first 64-bit client operating system to be released by Microsoft. The 64-bit version of Windows XP requires a computer with an Itanium processor, and is designed to take advantage of performance offered by the 64-bit processor. The hardware requirements for Windows XP 64-bit edition are different from the hardware requirements of a standard version of Windows XP Professional.

Component Minimum Requirement

Processor Intel Pentium (or compatible)
233MHz or higher
Intel Pentium II (or compatible)
300MHz or higher
Memory 64MB or 128MB
Video adapter and monitor with
SVGA resolution or higher
Peripheral devices like Keyboard, mouse, or other pointing device
Removable storage
CD-ROM or DVD-ROM drive if installing from CD
12x or faster CD-ROM or DVD-ROM

File Systems Supported

Windows XP Professional supports three file systems:

_ File Allocation Table (FAT16)
_ FAT32
_ New Technology File System (NTFS)

While Windows XP Home Edition adds a great deal to the feature set of Windows 2000, Windows XP Professional takes the product to the next level.

Many of the neat things that are part of Windows 2000 Professional are excluded from the Home Edition, but they are included in Win XP Professional. These features include the following:

_ IntelliMirror technologies

_ Group Policy functionality

_ Encrypting file system support

_ Multiprocessor support

Windows Vista

Windows Vista is a successor of Windows XP specially designed for business and professional users. Vista comes in 5 editions - three are designed for varying levels of personal users and two for business computing needs.

Hardware Requirement of Vista

Pentium 800 MHz Processor, 512 MB RAM, 15 GB free Hard disk space, VGA or Higher display, keyboard, mouse and NIC. Recommended Hardware Requirement P-IV 1 GHz or higher, 1GB RAM

Windows Vista Home Basic

This edition is designed for home users with most basic computing needs such as internet and e-mail access. It includes following features

1] Integrated desktop search makes it easy to find files, folders, messages, programs.

2] Windows Internet Explorer 7 with multi page browsing and enhanced security.

3] Sleep Mode - a fast-acting hibernation function that shut down or starts up your PC in seconds.

4] Windows side bar displays Gadgets that make information such as news photos, weather etc.

5] Windows calendar provides personal and shared schedules and task tracking.

6] Windows photo gallery and personal and public photo folders make it easy to view, label, organize and share digital images.

Windows Vista Ultimate

This edition is designed for home users who wants to do all. It includes all the features of Vista home premium plus additional features as follows

1] Windows Movie Maker

2] Windows Bit Locker Drive Encryption

3] Advance networking capabilities allows to join domain as well as provide group policy support.

Windows Vista Business

It is designed for small, mid-sized, and large business. This edition includes all features of Vista Home premium plus the following

1] Windows Mobility centre provides access to all settings you might want to adjust when using mobile PC.

2] Windows meeting space.

Other features of Windows Vista are as follows :-

1. Windows Vista Images Are Bigger

With Windows XP and Windows 2000, it was possible to create images that would fit easily on a single CD (less than 700MB). Even organizations that added applications, drivers, and utilities to their image typically ended up with an operating system image in the 1GB to 3GB range.

With Windows Vista, image size begins at about 2GB-compressed. Once this image is deployed, the size is often around 5GB or more, and there's no way to reduce it. If you add additional applications, drivers, or other files, this image obviously grows even larger.

So how will you deploy the image? Does your network have the necessary capacity? (10MB networks or non-switched networks are not sufficient.) If you want to use CDs, how many can you deal with? You'll need three or four. DVDs (with a capacity of 4.7GB each) are now easy to create, so you can deploy using DVD drives if you have them. (If not, consider adding DVD drives when buying the next round of PCs.)

With USB memory keys growing in size (as large as 4GB or more) and shrinking in price, it would be quite easy to use one for deploying Windows Vista, since you can make a bootable key as long as the computer's BIOS supports it.

Finally (though this doesn't relate to image size), take note that there is no longer an I386 directory. Instead, all components, whether installed or not, reside in the Windows directory (although not in the standard SYSTEM32 directory). When installing a new component, the necessary files will be pulled from this location.

2. Security is Enhanced

A number of Windows Vista security enhancements will impact deployment. For example, configuring Windows Vista to support "low rights" users, where the logged-on user does not have administrator rights, is easier. Some applications failed to work on Windows XP when users did not have administrator access because they assumed they would have full access to the C: drive and all parts of the registry. With Windows Vista, applications that attempt to write to restricted areas will have those writes transparently redirected to other locations in the user's profile.

The second big change here is that non-administrators can load drivers. This lets users attach new devices without needing to call the help desk in tears.

The third difference you'll find is that Internet Explorer® can automatically install ActiveX® controls using elevated rights. A new service can perform these installations on the user's behalf (if, of course, the IT administrator allows this via Group Policy).

Some of you may currently be using Power User rights on Windows XP, but this really does not offer many benefits (in terms of restricting user rights) over simply granting full Administrator privileges. Because of this, the Power Users group in Windows Vista has been removed, although it can be put back if required using a separate security template that can be applied to an installation of Windows Vista.

Sometimes you will need administrator rights, but this doesn't mean you want to run with admin rights all the time. So Windows Vista adds User Access Control (UAC), which causes most user applications-even for Administrators-to run with restricted rights. For applications that require additional rights, UAC will prompt for permission, asking either for permission to run with elevated privileges or for other user credentials that can replace the logged-on users.

There are also enhancements to the firewall built into Windows Vista. The new firewall can now control both inbound and outbound traffic, while still being fully configurable via Group Policy.

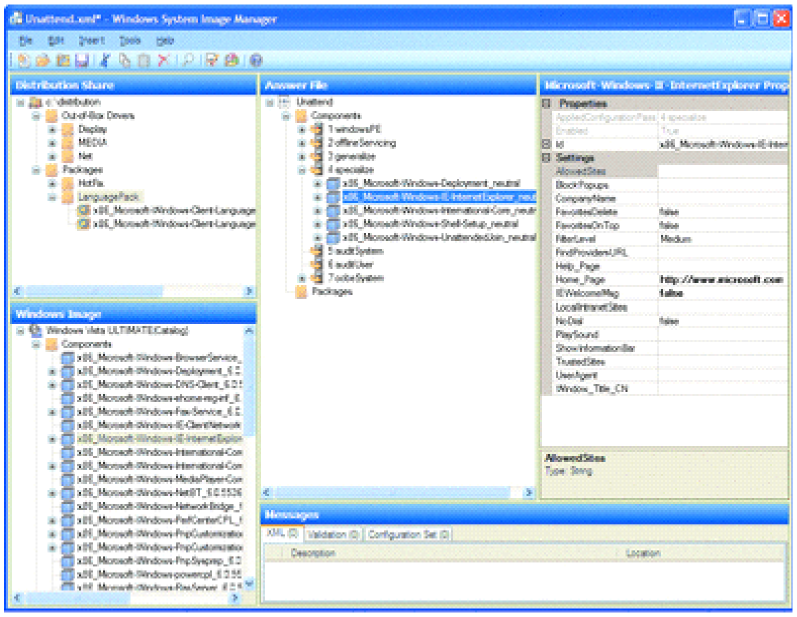
Finally, BitLocker™ full-volume encryption, which is included with Windows Vista Enterprise and Ultimate, allows the entire operating system volume to be encrypted. The volume can then be read only from within Windows Vista and only when the right keys are provided, either from the computer's built-in Trusted Platform Module (TPM) 1.2 chip, a USB key, or typed into the keyboard. (Note that only TPM 1.2 or later is supported.)

3. Windows Vista Is Componentized

One of the biggest architectural changes in Windows Vista is that it is now a completely componentized operating system. This affects deployment in the following ways.

Configuring which Windows Vista features should be installed requires configuring the components to be enabled. New tools, like the Windows System Image Manager, shown in Figure 1, assist with this.

Security updates, language packs, and service packs are simply components. Tools such as Package Manager (PKGMGR) can be used to apply these to Windows Vista.



Microsoft-Windows-IE-InternetExplorer Properties

Properties	
AppliedConfigurationPass	4 specialize
Enabled	True
Id	x86_Microsoft-Windows-IE-Inter
Settings	
AllowedSites	
BlockPopups	
CompanyName	
FavoritesDelete	false
FavoritesOnTop	false
FilterLevel	Medium
FindProviderURL	
Help_Page	
Home_Page	http://www.microsoft.com
IEWelcomeMsg	false
LocalIntranetSites	
NoDial	false
PlaySound	
ShowInformationBar	
TrustedSites	
UserAgent	
Window_Title_CN	

AllowedSites
Type: String

Figure 1 : **Windows System Image Manager (Click the image for a larger view)**

In addition, all servicing can be performed offline or online. You can even apply changes to Windows Vista or a Windows Vista image when Windows Vista is not currently running. This is ideal for deployments: the operating system can be patched before it boots onto your network for the first time.

Drivers are also treated as components, so they can be added and removed easily-even offline. This means you can add drivers to existing images, even just-in-time (as the machine boots for the first time) during the deployment process. And this applies to mass-storage drivers as well; no longer do you need to create a new image just to add a new mass storage driver.

Windows Vista exposes more settings, with most components providing configurable options, so it's easier to set installation defaults that can be managed on an ongoing basis using Group Policy. For a rundown of new tools in Windows Vista, see the sidebar "Tools You Need; Tools to Forget."

4. Text-Mode Installation Is Gone

The basic process used to install Windows XP has been unchanged since the earliest days of Windows NT®. This time-consuming procedure involved an initial text-mode installation step in which every operating system file was decompressed and installed, all registry entries were created, and all security was applied. Now with Windows Vista, this text-mode installation phase is completely gone. Instead, a new setup program performs the installation, applying a Windows Vista image to a computer.

Once this image is applied, it needs to be customized for the computer. This customization takes the place of what was called mini-setup in Windows XP and Windows 2000. The purpose is the same: the operating system picks the necessary settings and personality for the specific computer it was deployed to.

The image preparation process has also changed. With Windows XP, you would "Sysprep" a machine to prepare the reference operating system for deployment. With Windows Vista, you'll still run Sysprep.exe (installed by default in C:\Windows\System32\Sysprep), which will "generalize" the machine for duplication.

Windows Vista (any version) is provided on the DVD as an already-installed, generalized (Sysprepped) image, ready to deploy to any machine. Some customers may choose to deploy this image as-is (possibly injecting fixes or drivers using the servicing capabilities described earlier).

5. Boot.ini Is History

That's right, the Boot.ini file is not used in Windows Vista or in the new Windows PE 2.0. Instead, a new boot loader, bootmgr, reads boot configuration data from a special file named BCD. A brand new tool called bcdedit.exe (or a separate Windows Management Instrumentation or WMI provider) is used to maintain the contents of the BCD. A Windows PE 2.0 boot image can be configured in BCD too, making it easy to boot into either Windows Vista or Windows PE without making any other changes to the machine. This flexibility can be useful in recovery or maintenance scenarios.

6. Settings Are Configured in XML

With Windows XP (and previous versions of Windows PE) configuration information was stored in various text files. These text files have been replaced with an XML file.

Unattend.txt, which was used to configure how Windows XP is installed, has been replaced by unattend.xml. Unattend.xml also replaces three other files:

- Sysprep.inf, which was used to configure how a Windows XP image is customized when deployed to a machine using a mini-setup.
- Wimbom.ini, which was used to configure Windows PE.
- Cmdlines.txt, which was used to specify a list of commands to execute during mini-setup.

An example of unattend.xml can be downloaded from TechNet Magazine at microsoft.com/technet/technetmag/code06.aspx.

You may still use separate files if you want, though. You don't need to put all configuration items in a single unattend.xml file. The high-level schema of the new XML configuration file is well defined, with each phase of the deployment process represented. The actual configuration items are specified on the appropriate operating system components and these items are dynamically discovered from the components themselves.

With Windows XP, most IT professionals used Notepad to edit the various configuration files. You can still do that, but the Windows System Image Manager tool I discussed earlier can be used to inspect the Windows Vista image, determine what settings are available, and allow you to configure each one.

Another tool to aid deployment is the User State Migration Tool (USMT) 3.0, which is expected to be released at the same time as Windows Vista. It will also use XML configuration files in place of the .inf files that were used in previous versions. See "Migrating to Windows Vista Through the User State Migration Tool" for more information.

7. No More HAL Complications

With Windows XP, technical restrictions prevented the creation of a single image that could be deployed to all computers. Different hardware abstraction layers (HALs) meant you had to maintain multiple images. (For more on this see the Knowledge Base article "HAL options after Windows XP or Windows Server 2003 Setup") Most organizations needed two or three images per platform (x86 and x64) and some chose to have even more-though each image brings added costs and complexity.

In Windows Vista, those technical restrictions are gone; the operating system is able to detect which HAL is required and automatically install it.

8. Windows PE Rules

Windows PE 2.0, the new version that will be released with Windows Vista, is a key part of the deployment process. Even the standard DVD-based installation of Windows Vista uses Windows PE 2.0, and most organizations will be using it (often customized for the organization's specific needs) as part of their deployment processes.

Compared to MS-DOS®-based deployment, Windows PE 2.0 brings numerous benefits, including less time spent trying to find 16-bit real-mode drivers. (It's not even possible to find these any more for some newer network cards and mass storage adapters.) Better performance from 32-bit and 64-bit networking stacks and tools, as well as large memory support are also advantages. And don't forget support for tools such as Windows Scripting Host, VBScript, and hypertext applications.

Windows PE has been available for a few years (the latest version, Windows PE 2005, was released at the same time as Windows XP SP2 and Windows Server 2003 SP1), but not all organizations could use it; it required that you have Software Assurance on your Windows desktop operating system licenses. With Windows PE 2.0, that's no longer the case. All organizations will be able to download Windows PE 2.0 from microsoft.com and use it freely for the purposes of deploying licensed copies of Windows Vista.

Like Windows Vista itself, Windows PE 2.0 is provided as an image that is componentized and can be serviced both online and off. As with Windows PE 2005, several optional components can be added, although Windows PE 2.0

includes some new ones: MSXML 3.0, Windows Recovery Environment, language packs, font packs, and so on. New tools like peimg.exe are provided for servicing Windows PE 2.0. Peimg.exe can also be used for adding drivers-including mass storage devices, which no longer require any special handling.

For more information on Windows PE 2.0, see Wes Miller's article in this issue of TechNet Magazine.

9. It's All about Images

With Windows XP, some companies used the image creation capabilities of the Systems Management Server (SMS) 2003 OS Deployment Feature Pack or third-party image creation tools. There was no generic image creation tool available from Microsoft. That's changed with Windows Vista: new tools have been created to support the Windows Imaging (WIM) file format. Unlike many other image formats, WIM images are file-based, enabling them to be applied to an existing partition non-destructively. This has great advantages in deployment processes, since user state can be saved locally instead of on a network server, eliminating what is frequently the largest source of network traffic during a deployment.

Because WIM files are file-based images, they (obviously) are not sector-based, so there are no issues around different-sized disks or partitions. A WIM image contains only the contents of a single disk volume or partition, so if you have multiple partitions to capture, you create a separate image for each one. But each of these images can be stored in the same WIM file, since the WIM file format supports multiple images per file.

The WIM file format also supports single-instance storage, so duplicate files (even from different images) are automatically removed. Between this and the advanced compression techniques employed, WIM images are typically smaller than images created by other tools. However, because of the extra processing, they do take longer to create. This size versus performance trade-off is fair enough; since you typically capture the image only once and then deploy it many times, the network traffic savings can be substantial.

The IMAGEX command-line tool interfaces with the lower-level WIMGAPI API (which is fully documented for use in custom tools too), and is used to create and manipulate WIM images. It also provides a mechanism for mounting a WIM image as a file system. Once mounted, the image can be read and modified using standard Windows tools since it looks like a normal removable media drive. This facility opens up whole new servicing opportunities.

10. Deployment Is Language-Neutral

Windows XP supported different languages in two ways. You could either deploy localized versions of Windows XP, requiring a different image for each language, or you could deploy an English Multilanguage User Interface (MUI) version with added language packs. There were advantages and disadvantages to each approach, but in most cases organizations that needed to support multiple languages took the MUI route, dealing with the limitations of running with an operating system that was effectively English at its core. Organizations that worked only with one language typically chose to use only the localized versions.

Now with Windows Vista, the entire operating system is language-neutral. One or more language packs are added to this language-neutral core to create the image that is deployed (although only some versions of Windows Vista support multiple languages).

Servicing of Windows Vista is also language-neutral, so in many cases only one security update is needed for all languages. And configuration is language-neutral, so one unattend.xml can be used for all language.

1.5 & 1.6 Check Your Progress.

State Whether the following statements are true or false

1] Disk Operating System (DOS) is a single user single-process operating system

2] DOS support Graphical User Interface.

3] In DOS First 640k is Conventional Memory

4] Windows 9X support GUI as well as Command Line Interface (CLI)

5] SYSTEM.DAT - Contains information about hardware and system settings.

6] Windows 9x supports long file names up to 155 characters

6] Windows XP support two processors.

7] Windows XP has two editions as 32bit and 64 bits.

8] Windows Vista is a successor of Windows 98.

9] Windows Vista supports 4 way SMP i.e. 4 processors.

10] Windows XP Home Edition can not take part in Domain Networks.

Fill in the Blanks

1] DOS stands for ----------------------------

2] Windows 9x supports long file names up to ---------- characters.

3] Windows XP supports maximum ----------- CPUs.

4] Windows Vista does not use ------------------ file as used in windows 2000 and windows XP.

5] Windows XP supports ---------------------, -----------------and ----------------- file systems.

Match the followings

1. Windows XP	1. Single user single task operating system.
2. DOS	2. Two editions of 32 and 64 bits.
3. Windows Vista	3. Desktop Operating System.
4. Windows 2000	4. Boot.ini file is not used. Professional

1.7 SUMMARY

Operating System can be divided into 2 groups i.e. single and multiprocess.

Multitasking is the capability of operating system to handle more than one task at a time.

Desktop as is single use operating system Networks Operating System is multiuser Operating System.

The main benefit of using XP include its reliability, performance, security, ease of use, support for remoter users, management and development capabilities. Windows Vista is also language neutral.

Source : *http://kurdprogram.com*

1.8 CHECK YOUR PROGRESS - ANSWERS

1.1 & 1.2

1] False 2] True 3] False 4] True

1.3 - 1.4

1] Single 2] Client 3] Multi 4] High

1.5 - 1.6

1]	True	2]	False	3] True
4]	True	5]	True	6] False
6]	True	7]	True	8] False
9]	False	10]	True	

Fill in the Blanks

1] Disk Operating System. 2] 255 3] Two 4] Boot.ini

5] FAT16, FAT 32 and NTFS

Match the followings

1-2 2-1 3-4 4-3

1.9 QUESTIONS FOR SELF - STUDY

1. What is mean by multitasking? Explain its types.
2. Explain features of Desktop Operating System.
3. Explain Features of Network Operating System.
4. Explain Dos System Files in detail.
5. Explain Windows qx core components.
6. Explain various features of Windows Vista.

1.10 SUGGESTED READINGS

Operating systems By Stuart E. Madnick, John J. Donovan

❑ ❑ ❑

Notes

Notes

Chapter : 2

Processor Management

2.0 OBJECTIVES

After studying this chapter you will be able to -

- Describe/Analyze processor management with the observance of Job Scheduler and peocess scheduler.
- Discuss the process synchronization and multi-processor systems.

2.1 INTRODUCTION

In most systems it is necessary to synchronize processes and jobs, a task that is also -performed by modules of processor management. The job scheduler can be viewed as a macroscheduler, choosing which jobs will run. The process scheduler can be viewed as a microscheduler, assigning processors to the processes associated with scheduled jobs.

The user views his job as a collection of tasks he wants the computer system to perform for him. The user may subdivide his job into job steps, which are sequentially processed sub tasks (e.g., compile, load, execute). The system creates processes to do the computation of the job steps.

Job scheduling is concerned with the management of jobs, and processor scheduling is concerned with the management of processes. In either case, the processor is the key resource.

In a non-multiprogramming system no distinction is made between process scheduling and job scheduling, as there is a one-to-one correspond-ence between a job and its process and only one job is allowed in the system at a time. In such a simple system the job scheduler chooses one job to run. Once it is chosen, a process is created and assigned to the processor.

For more general multiprogramming systems, the job scheduler chooses a small subset of the jobs submitted and lets them "into" the system. That is, the job scheduler creates processes for these jobs and assigns the processes some resources. It must decide, for example, which two or three jobs of the 100 submitted will have any resources assigned to them. The process scheduler decides which of the processes within this subset will be assigned a processor, at what time, and for how long.

2.2 STATE MODEL

Let us pretend we are a job and follow our path through the states of Figure 2.1.

My friendly programmer wrote my program code several days ago. He submitted my code and appropriate control cards to the computing centre (submit state). An operator took my code and caused it to be read into the computer and copied onto a disk where a job scheduler could examine it (hold state). Eventually, the job scheduler chose me to run and set up a process for me (ready state). To get my process into a ready state faster on a manually scheduled system, my friendly pro-grammer might have inserted a green card (a five-dollar bill) in front of my deck.

After being scheduled and set up by the job scheduler, my process was finally assigned a processor to execute my code (running state) by the process scheduler module. At that time I was given a "time quantum," i.e., a time limit for running. When that time quantum was up, I still was not completed. Therefore, my process was sent back into the ready state.

Sometime later the process scheduler gave my process another time quantum and assigned a processor (running state). During this time quantum my process almost finished but still had to print my answer. Therefore, my process initiated another process, an I/O process, and had to wait (blocked state) until my I/O was completed.

My I/O process has a similar state diagram-it became ready when my CPU process initiated I/O. When the I/O process finished, it signalled my blocked process that it was through.

When the operating system received the signal that my I/O process was finished, the traffic controller module changed my CPU process from the blocked state to the ready state again. This time when the process scheduler assigned a processor to my process (running state), I finished all my work (completed state).

We can identify the following modules of processor management.

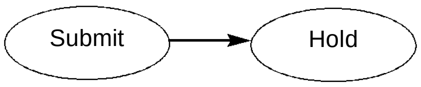
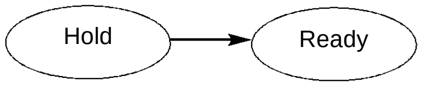
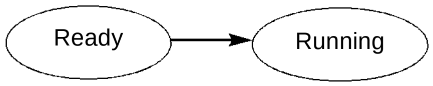
MODULE	FUNCTION
Spooler	1 Place all submitted jobs into a form for processing by the job scheduler 2. Keep track of all jobs in system 3. Select a job to run and create corresponding process
Job scheduler	
Process scheduling	4. Select a process to run and allocate a processor
Traffic controller	5. Keep track of status of all processes 6. Provide the mechanics for changing Process' states  7. Coordinate interprocess synchronization and communication 

Figure 2.1 Job Scheduling

We have shown the process view of an operating system in Figure 2.1, noting the scope of job scheduling.

2.2.1 Job Scheduling

The job scheduler is the "super" manager, which must:

1. Keep track of the status of all jobs. It must note which jobs are trying to get some service (hold state) and the status of all jobs being serviced (ready, running, or blocked state).
2. Choose the policy by which jobs will "enter the system" (i.e. go from hold state to ready state). This decision may be based on such characteristics as priority,

resources requested, or system balance.

3. Allocate the necessary resources for the scheduled job by use of memory, device, and processor management.
4. Deallocate these resources when the job is done.

2.2.2 Process Scheduling

Once the job scheduler has moved a job from hold to ready, it creates one or more processes for this job. Who decides which processes in the system get a processor, when, and for how long? The process scheduling modules make these decisions.

Specifically, the following functions must be performed :

1. Keeping track of the status of the process (all processes are either running, ready, or blocked). The module that performs this function has been called the traffic controller
2. Deciding which process gets a processor and for how long. The processor scheduler performs this
3. Allocation of a processor to a process. This requires resetting of processor registers to correspond to the process' correct state. The traffic controller performs this task.
4. Deallocation of a processor, such as when the running process exceeds its. Current quantum or must wait for an I/O completion. This requires that all processor state registers be saved to allow future reallocation. The traffic controller performs this task.

2.2.3 Job and Process Synchronization

On the job level there are usually mechanisms for passing conditions between jobs. For example, we might want to prevent job step 4 from being performed if job step 1 had failed.

On the process level there must be mechanisms to prevent race conditions. A race condition exists when the result of a computation varies, depending on the timing of other processes. For example, one process requests a printer while another process is printing; if the printer were to be also assigned to the second process, the output would be intermixed between the two processes. We must have a mechanism to stop the second process from running until the first process releases the printer.

The P and V operators and semaphores are one set of mechanisms for coordinating the assignment of processors to processes. In addition, we must be aware of possible deadlock situations in cases where there are two processes, each of which is waiting for resources that the other has and will not give up. In this situation, no processor can be assigned to either process.

2.2.4 Structure of Processor Management

The three main sections of this chapter, job scheduling, process scheduling, and synchronization, are all concerned with the assignment of processors to processes. Hence, they are all included in this chapter on processor management. In some cases, especially with job scheduling, interaction with the other resource managers takes place.

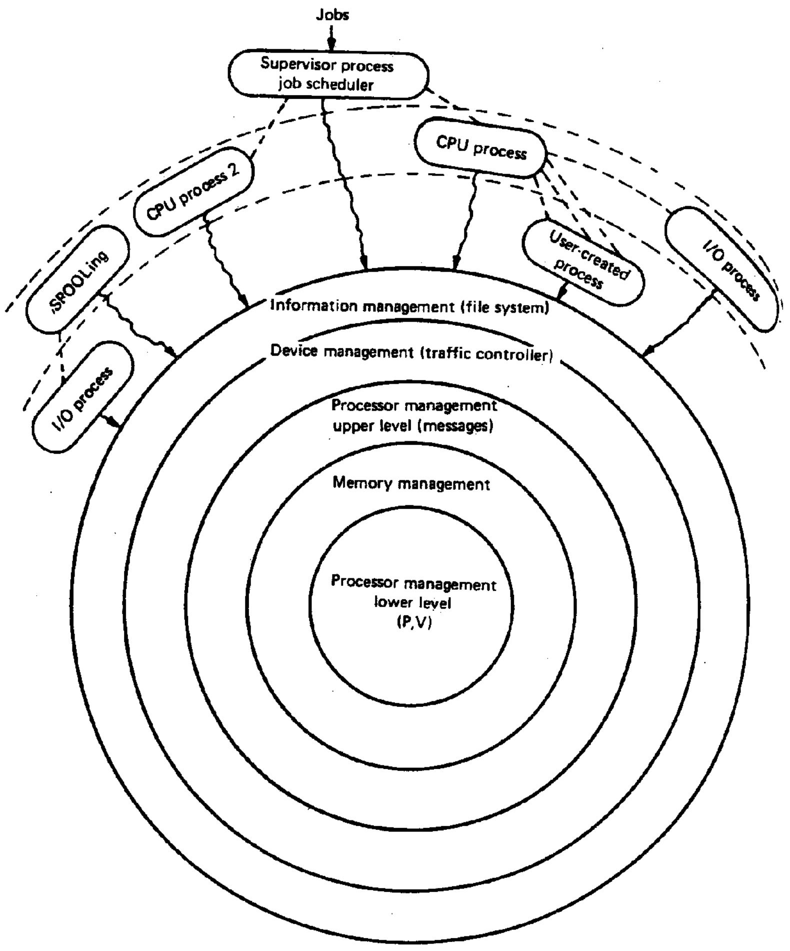

Jobs
Supervisor process job scheduler
CPU process
CPU process 2
SPOOLing
User-created process
I/O process
I/O process
Information management (file system)
Device management (traffic controller)
Processor management upper level (messages)
Memory management
Processor management lower level (P,V)

Figure 2.2 Hierarchical view of computer-based operating system levels for a real or extended machine.

In summary, processor management operates on two levels-assigning processors to jobs and assigning processors to processes.

On the job level, processor management is concerned with such questions as which jobs will be run and which will run first. At this first level processor management is not concerned with multiprogramming. It assumes that once a job is scheduled, it will run. Job scheduling can run concurrently with other users' pro-grams or other system functions. Thus job scheduling may itself be a separate process, as shown in Figure. 2.1

Once a job is scheduled, the system must perform the functions of creating processes, destroying processes, and sending messages between processes. In general system users' processes, job scheduler, and other processes may request these functions. Thus the functions may be common to all address spaces (see Figure 2.2 Processor Management Upper Level).

In a multiprogramming environment the process scheduler and the synchronization mechanisms may be called by all modules of the system. Thus they form the very centre of the kernel of a system (Figure 2.2 Processor Management Lower Level).

The job scheduler is like the coordinator of a contest; he decides who will enter the contest but not who will win it. The process scheduler decides which participant will win. In a nonmultiprogramming environment only one contestant is allowed to enter.

2.2 Check Your Progress.

Fill in the Blanks.

1. When the operating system received the signal that my I/O process was finished, the traffic controller module changed my CPU ______________

2. Allocate the necessary resources for the scheduled job by use of memory, device, and processor management ______________

2.3 JOB SCHEDULING

The functions of the job scheduler 'may be done quite simply, quite elaborately, or dynamically. In a timesharing system such as the Compatible Time Sharing System (CTSS), the job scheduling policy may consist of admitting the first 30 users that log in (i.e. arrive). A simple two-level priority algorithm allows a user with a higher priority to force the logout (Le. termination) of a lower priority user. The user to be logged out is chosen on the basis of the processor time he has used: the one who has used the most processor time gets logged out.

The job-scheduling algorithm of a batch system such as OS/VS-2 can be sophis-ticated. It takes into account not only the time a job arrives but also priority, memory needs, device needs, processor needs, and system balance.

In this section we discuss some simple job scheduling policies. We focus on the policies of job scheduling and their implications, dealing first with system without multiprogramming, then with more general multiprogramming systems.

2.3.1 Functions

The job scheduler can be viewed as an overall supervisor that assigns system resources to certain jobs. Therefore, it must keep track of the jobs, invoke policies for deciding which jobs get resources, and allocate and deallocate the resources.

We have chosen not to spend a great deal of time on extremely detailed mechanisms for performing the functions of job scheduling. We are mainly concerned here with the factors that go into scheduling, and the consequences that result. For example, if job scheduling is done by reference to memory needs only, what is the effect on turnaround time as compared with other scheduling policies?

Before discussing different policies, let us briefly address ourselves to the function of keeping track of jobs in a system. One mechanism for keeping track of jobs is to have a separate lob Control Block (lCB) for each job in the system. When a process is placed in hold state (Figure 2.3) a JCB is created for it with entries regarding its status and position in a job queue.

Job Identification
Current State
Priority
Time estimates
Etc.

Figure 2.3 Job Control Block

Figure 2.3 illustrates the type of information that may be contained in a job Control Block. Some information, such as priority and time estimate, is obtained from the job control cards submitted with the job. Other information, such as current state, is set by the operating system.

2.3.2 Policies

The job scheduler must choose from among the "hold" jobs those that will be made "ready" to run. In a small computing centre an operator may do this function. He may choose arbitrarily, choose his friends, or choose the shortest job. In larger systems, e.g., OS/360, all submitted jobs might be first stored on a secondary storage device whereupon the job scheduler can examine all such jobs. It can then choose which jobs will have system resources committed to them and hence will be run.

We should note the following :

The key concept is that there are more jobs that wish to run than can be efficiently satisfied by the resources of the system.

In particular :

i. most resources are finite (e.g., tapes, memory)

ii. many resources cannot be shared or easily reassigned to another process.

Scheduling is considered a. policy issue because its goals are usually subjective (and sometimes contradictory). For example, how would you rank the following goals?

1. Running as many jobs as possible per day (only run short jobs)
2. Keeping the processor "busy" (only run computation- intensive long jobs)
   Fairness to all jobs.

Typical considerations one must deal with in determining a job scheduling policy are :

3. Availability of special limited resources (e.g., tapes)

a. If you give preference, someone can "Cheat." For example, if you always first run the job that requests a plotter, then some users will always request a plotter.

b. If you don't give preference, some user suffers an extra delay (e.g., must wait for tape to be mounted or plotter adjusted after having waited to be. scheduled).

4. Cost-higher rates for faster service
5. System commitments-processor time and memory-; - the more you want, the longer you wait.
6. System balancing-mixing I/O-intensive and CPU- intensive.
7. Guaranteed service-setting a specific waiting time limit (I hour) or a general
8. Limit (within 24 hours). (At least don't lose the job!)
9. Completing the job by or at a specific time (e.g., 5 PM).

Once the job scheduler has selected a collection of jobs to run, the process scheduler attempts to handle the microscopic scheduling (dynamic assignment of processor to processes). Even ignoring time quanta limits, a process will typically run less than 100 ms before becoming blocked to await I/O completion or a similar event.

The job and process scheduler may interact. The process scheduler may choose to "postpone" ("roll out") a process and require that it go through the macro-level job scheduling again in order to complete. This is especially true in a timesharing system.

2.3 Check Your Progress.

Fill in the Blanks.

1. Once the job scheduler has selected a collection of jobs to run, the process scheduler attempts to handle the _____________.
2. The process scheduler may choose to _____________ a process.

2.4 PROCESS SCHEDULING

Once the job scheduler has selected a collection of jobs to run the. Process scheduler attempts to handle the microscopic scheduling (i.e. the dynamic assignment of processor to process). The process scheduler is also called the dispatcher or low-level scheduler in the literature. In a multiprogrammed environ-ment, a process typically uses the processor for only 100 ms or less before becoming blocked to wait for I/O completion or some other event.

The job and process scheduler may interact. The process scheduler may choose to postpone, or roll out, a process and require that it go through the macro-level job scheduling again in order to complete. This is especially true in a timesharing system.

In Figure 2.4 we see a single processor multiprogramming system. Processes A and D both request the use of a tape drive during execution. Let us assume there is only one tape drive available, and process a requests it first. The drive is assigned to process A. Later, when process D requests a tape drive, it must be blocked.

Being blocked is an attribute of the process, not the processor. It means that the processor will switch only among processes A, B, and C, and ignore process D. Usually process D will tie up a portion of memory. But in some systems it can be rolled out or paged out of memory onto secondary storage. In such a case, the blocking of process D has a logical effect that results in a redistribution of resources; the resources need not be wasted.

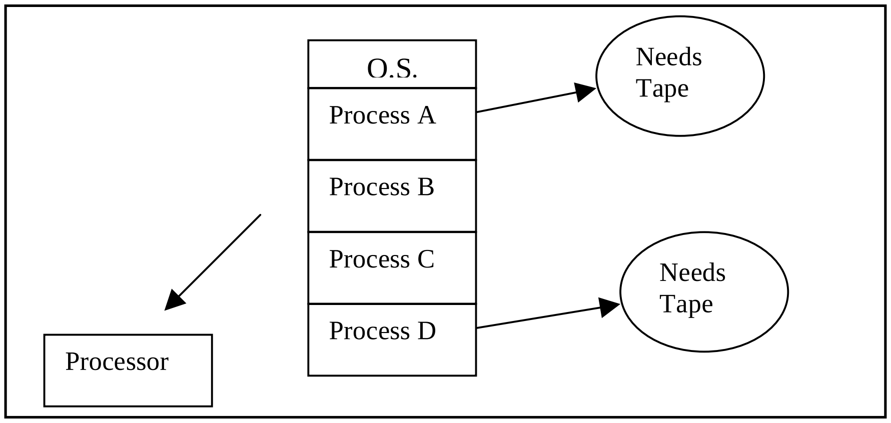


Fig 2.4 - Single processor multiprogramming system

2.4.1 Functions

The process scheduler must perform the following functions:

1. Keep track of the state of processes.
2. Decide, which process gets a processor, when, and for how long.
3. Allocate processors to processes.
4. Deanocate processors from processes.

Let us briefly discuss the function of keeping track of the status of the process. The module of the processor scheduler that performs these functions is called the traffic controller. One way of keeping track of the status of a process is to use a database associated with each process in the system called a Process Control Block (PCB). There is a separate PCB for each process; in fact, the PCB is the only really tangible part of a process (see Figure 2.5).

Process identification
Current state
Priority
Copy of active registers
Pointer to list of other
Processes in same state
Etc.

Fig 2.5 - Process Control Block (PCB)

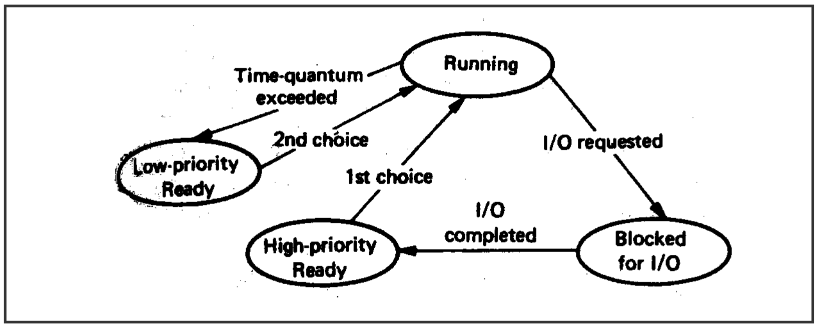


Figure 2.6 - A set of scheduling state transitions

Frequently an PCBs in the same state (e.g., ready, blocked, etc.) are linked together; the resulting list is often given a special name such as ready list. The traffic controller is caned whenever the status of a resource is changed. The block list may be further subdivided, one chain given for each reason that a process is blocked. Thus if a process requests a device and their device is already in use, the process is linked to the block chain associated with that device.

When the device is released, the traffic controller checks its block chain to see if any processes are waiting for that device. If so, these processes are placed back in ready state to rerequest the device. Alternatively, one of the processes may be assigned the device and placed in ready state, while all the other processes waiting for that device remain blocked.

2.4.2 Policies

The scheduling policy must decide which process is to be assigned a processor and for how long. The length of time a process is assigned a processor may depend on one, or some combination of, the following events :

The process is complete.

The process becomes blocked.

A higher priority process needs the processor.

A time quantum has elapsed.

An error occurs.

Scanning the PCB ready list and applying some policy make the decision as to which process is to run. The ready list can be organized in two ways:

1. Each time a process is put on the ready list (from hold, running, or blocked) it is put in its "correct" priority position in order. Then, when the processor becomes free, the top process on the list is run.

2 Processes are arbitrarily placed on the ready list. When it is necessary to find a process to run, the scheduler must scan the entire list of ready processes and pick one to run.

Typical process scheduling policies include the following :

1. **Round robin:** Each process in turn is run to a time quantum limit, such as 100 ms.
2. **Inverse of remainder of quantum:** If the process used its entire quantum last time, it goes to the end of the list. If it used only half (due to I/O wait), it goes to the middle. Besides being "fair," this works nicely for I/O-bound jobs.
3. **Multiple-level feedback variant on round robin:** When a new process is entered, it is run for as many time quantums as all other jobs in the system. Then regular round robin is run.
4. **Priorities :** The highest priority ready job is selected. Priority may be purchased or assigned (e.g., a nuclear reactor process may be run first).
5. **Limited round robin:** Jobs are run round robin until some fixed number of times. Then they are run only if there are no other jobs in the system.
6. **System balance :** In order to keep the I/O devices "busy," processes that do a lot of I/O are given preference.
7. Preferred treatment to "interactive" jobs. If a user is directly communicating with his process (i.e., an interactive process), the process is given the processor immediately after user input to provide rapid service.
8. **Merits of the job :** In some cases it is advantageous for the system itself to assign priorities. For instance, it may automatically assign high priorities to short jobs. Another example would be for the system to designate priorities so as to achieve a balanced load, such as to have one job with heavy input/ output requirements (I/O-bound) and one job with very little input/output (CPU-bound) running concurrently. If all jobs are at one extreme, either the I/O channel will be overworked or the CPU idle, or vice versa. For maximal throughput it is thus advantageous for the system to assign priorities so as to maintain the proper mix and balanced load. Similarly, the system may desire to increase the priority of a job that is tying up resources (such as previously scheduled tape drives) in order to complete the job and release these resources for other use.

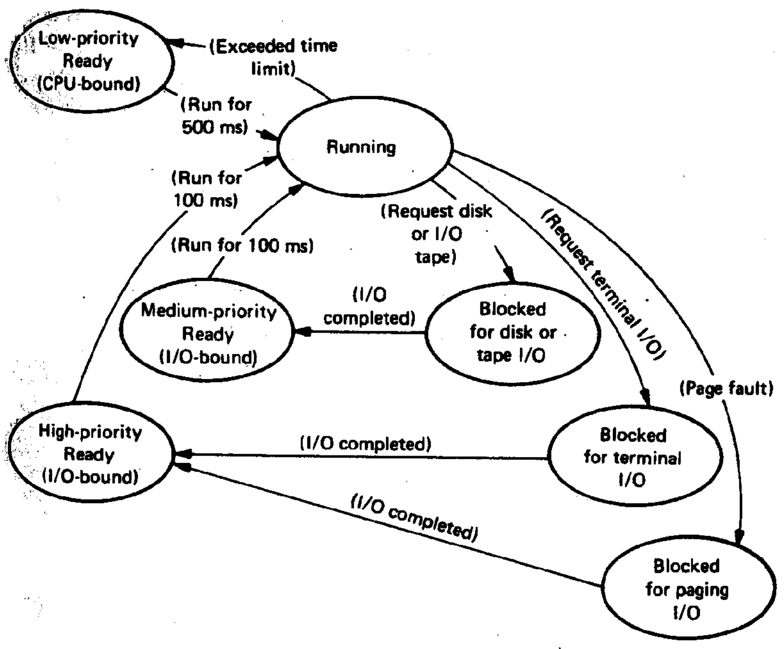


Figure 2.7 Process Scheduling

As in almost any rationing situation, there is considerable controversy over what are equitable, efficient, or economical policies. The reader may wish to consult some of the references for additional details of scheduling policy; a considerable amount of computer folklore has developed in this area.

2.4.3 Process State Diagrams for Scheduling

It is often convenient and graphically clearer to use the process state diagram to help explain the process-scheduling algorithm, as demonstrated in Figure.2.6 In this example, two ready states (low-priority ready and high-priority ready) are indicated. A process enters low-priority ready if it exceeds its time quantum while running, and high-priority ready when it moves from blocked to ready state.

A possible scheduler policy may be:

1. To select a process to run from the high-priority ready list
2. If there are no processes in high-priority ready, to select a process from the low-priority ready list.

The state transitions of Figure 2.7 have the effect of characterizing ready processes as either I/O-intensive (high-priority ready) or CPU-intensive (low priority ready). The scheduler policy gives preference to I/O-intensive processes.

2.4 **Check Your Progress.**

Fill in the Blanks.

1. In _____________ each process in turn is run to a time quantum limit, such as 100 ms.
2. In _____________ if the process used its entire quantum last time, it goes to the end of the list. If it used only half (due to I/O wait), it goes to the middle.Besides being "fair," this works nicely for I/O-bound jobs.

2.5 MULTIPROCESSOR SYSTEM

To enhance throughput, reliability, computing power, parallelism, and economies of scale, additional processors can be added to some systems. In early multiprocessor systems the additional processors had specialized functions, e.g., I/O channels. Later multiprocessing systems evolved to include the concept of one large CPU and several peripheral processors (CDC 6600). These processors may perform quite sophisticated tasks, such as running a display. A more common type of multiprocessing is a system having two or more processors, each of equal power, e.g., HIS 6180, IBM 158MP and 168MP, UNIVAC 1108, and Burroughs 6700. There is also the computer network, in which many different computers are connected, often at great distances from one another. Let us focus on the multi-processor system with many CPUs, each of equal power.

There are various ways to connect and operate a multiprocessor system, such as (1) separate systems, (2) coordi-nated job scheduling (3) master/slave scheduling and (4) homogeneous scheduling. These techniques differ, in degree of scheduling sophistication; each will be discussed in turn.

2.5.1 Separate Systems

Some systems, for example, the IBM System/360 Model 67, can be logically subdivided into two or more separate systems, each with one processor, some main memory, and peripheral devices. This is just like having two or more .separate computing systems, all in the same room.

The advantage to this organization is that processors, memories, and I/O devices can be easily, though manually, switched. For example, the ensemble can be con-figured to be two systems, each with one processor, 512K bytes of memory, and 8 disk drives, or one system with one processor, 1024K bytes of memory, and 16 disk drives (second processor is unused); other such alternatives exist.

This configuration flexibility is useful if there are some jobs that require the full complement of memory and/or I/O device resources. Alternatively, if one processor is being repaired, all the other resources can be pooled into one large system rather than be allowed to sit idle.

In this separate system situation, there is no job or process scheduling between the processors except that which is accomplished manually.

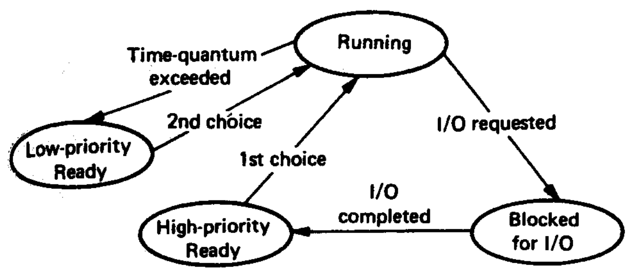


Figure 2.8 - A set of scheduling state transitions

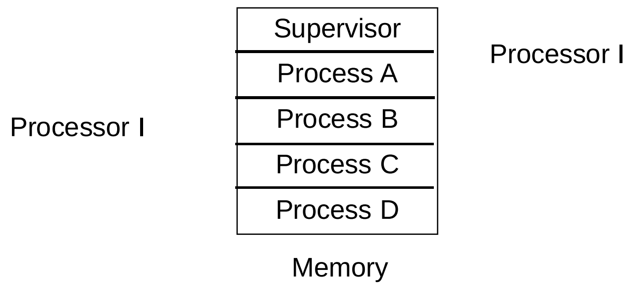


Figure 2.9 - Multiprogramming with multiple processors

2.5.2 Master/Slave Scheduling

The permanent assignment of a job to a system, as in the coordinated job scheduling technique, cannot handle the short-term balancing of resource demand. In a general multiprocessor system all memory and I/O devices are accessible to all processors. Memory and I/O devices are assigned to the processes, not to the processors. The processors are assigned to exe-cute processes as determined by the process scheduler-except that there are now multiple processors available for assignment.

In a master/slave scheduling technique, one processor maintains the status of all processes of the system and schedules the work of all the slave processors.

2.5.3 Coordinated Job Scheduling

A variation on the separate system multiprocessor technique is the coordinated job scheduling approach (this is also called loosely coupled multiprocessing). Each processor is associated with a separate system. When a job arrives, it may be assigned to any system. The assignment of a job to a system may be based on a variety of requirements and policies, such as assigning the job to the system with the lightest load.

To accomplish this balancing, all job scheduling must be coordinated. This may be done manually, by a special-purpose computer (as done on the Octopus system at the Lawrence Radiation Laboratory), or by one of the actual processing systems (as done on 18M's OS/VS-2 Job Entry System).

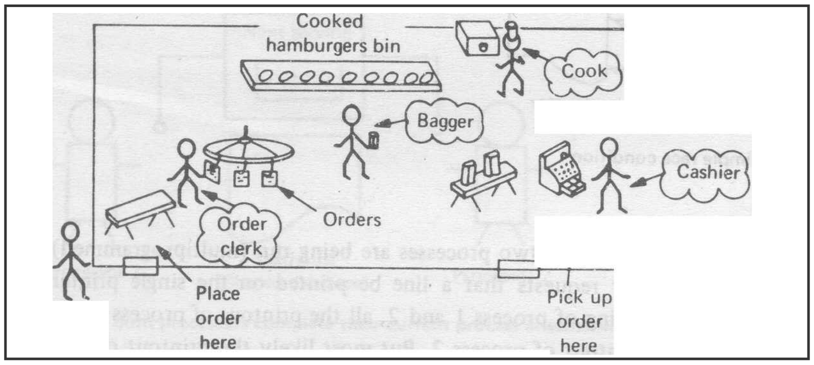


Figure 2.10 Operation of a hamburger stand

This simple example illustrates forms of communication and cooperation that we frequently take for granted. The bagger can fill orders only when there are orders to be processed. What does he do if there is a slack period and the order list is empty? He goes to sleep: How is he awakened when a customer arrives and places an order? The order clerk kicks him! It is a useful exercise to look for other necessary forms of communication and cooperation. For example, what prevents the cook from burying the stand under a pile of hamburgers 100 feet deep during a long slack period?

Analogous situations exist in a computing system among the processes. In some cases the coordination is forced upon them by the operating system because of the scarcity of resources, for example, the need to wait for access to an I/O channel. In other cases, a single job may consist of several processes that all work together. An information retrieval system, such as an airline reservation system, may have several processes performing separate tasks, as in our hamburger stand example. One task may accept requests, check for input correctness, and pass the request along to the "worker" process. The worker process accesses the data to find the necessary information and sends it to the "output" process. The output process routes the response back to the original requester. Associated with processor allocation. and interprocess communications are two synchronization problems, race condition and deadly embrace.

2.5 Check Your Progress.

Fill in the Blanks.

1. This configuration flexibility is useful if there are some jobs that require the full complement of memory and/or I/O ___________
2. A variation on the separate system multiprocessor technique is the ______________ approach (this is also called loosely coupled multiprocessing).

2.6 PROCESS SYNCHRONIZATION

The problem of process synchronization arises from the need to share resources in a computing system. This sharing requires coordination and cooperation to ensure correct operation.

An example may illustrate some of these points. Figure 2.10 illustrates a typical rapid-service hamburger stand. You submit your order to the order clerk. He places your order on a turntable. The bagger takes orders off the turntable, one at a time, and puts the requested items into a bag. He then delivers the completed order to the cashier. The cashier processes these bags, one at a time, receiving money from you and giving you your order. Furthermore, there is a cook working in the back room, continually replenishing the supply of cooked hamburgers used by the bagger. In this example, each process (person) operates largely independently. The only communication is via certain shared areas: the order list turntable, the cooked hamburger bin, and the checkout counter.

It is necessary to establish good coordination and communication among pro-cessors in a decentralized operating system.

In Figure 2.11 we assume that two processes are being run (multiprogrammed). Each process occasionally requests that a line be printed on the single printer. Depending on the scheduling of process 1 and 2, all the printout of process I may precede or follow the printout of process 2. But most likely the printout of each process will be interspersed on the printed paper.

One solution to this predicament is to require that a process explicitly request use of the shared resource (the printer in this case) prior to using it. When all use of the resource is completed (e.g., all printout completed), the resource may then be released by the process. The operations request and release are usually part of the operating system facilities handled by the traffic controller. If a process requests a resource that is already in use, the process automatically becomes blocked. When the resource becomes available as a result of a later.

In addition to physical devices, there are other shared resources, such as common tables, that require the same type of process synchronization.

For an example of such a race condition, consider Figure 2.12, which depicts a procedure for selecting a process from the ready list and placing it in the running state. This procedure is executed by the process scheduler. This algorithm works well for a single processor, but runs into trouble for multiple processors, as shown in Figure 2.12.

Assume that two processes, each on a separate processor, go blocked at approximately the same time. After saving the status of the current processes, each processor determines next process to be served. Since both processors use the same algorithm independently, they will choose the same process. Furthermore, in readjusting the selection criterion, it is possible to upset the ready list.

2.6.1 Race Condition

A race condition occurs when the scheduling of two processes is so critical that the various orders of scheduling them result in different computations. Race conditions result from the explicit or implicit sharing of data or resources among two or more processes. Figure 2.11 illustrates a simple form of race.

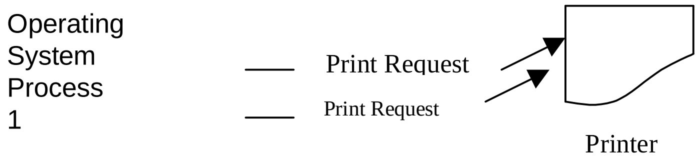
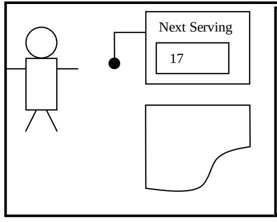


Figure 2.11 Simple race condition

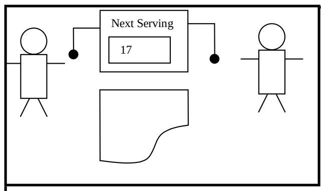
2.6.2 Synchronization Mechanisms

Various synchronization mechanisms are available to provide interprocess co-ordination and communication. In this section several of the most common tech-niques will be briefly presented; the references provide more detail and additional alternatives.



When current process is completed or postponed:

1. current process in ready list
2. Mark it "complete" or "postponed"
3. Note "next serving" number (17)
4. Find processes 17 in ready list
5. Make note of process to be done
6. Adjust ready list for "next serving" number (18)



Assume both processors complete their current process at same time:

1. Processor 1 finds its current process in ready list
2. Processor 2 finds its current process in ready list
3. Processor 1 marks its current process
4. Processor 2 marks its current process
5. Processor 1 notes "next serving" (17)
6. Processor 2 notes "next serving" (17)
7. Processor 1 finds process 17 in ready list and notes it
8. Processor 2 finds process 17 in ready list and notes it
9. Processor 1 adjusts ready list "next serving" (18)
10. Processor 2 adjusts ready list "next serving" (19)

Figure 2.12 Scheduling (multiple processors)

Now both processors are simultaneously working on the same process and process 18 has been skipped.

Set (TS) instruction that performs both steps 1 and 2 as an indivisible operation. There are similar instructions available on most contemporary computers. The reader should convince himself that the lock and unlock operations do, in fact, prevent race conditions. For example, consider the three processes has been skipped.

```
Lock (X)              UnLock (X)
 TS X                  MVI  X,00
          BNZ *- 4
```

PROCESS 1	PROCESS 2	PROCESS 3
Lock (XI	LOCK (XI	LOC (XI
(critical : database	: 1...1	: (...)
manipuLationsl		
UNLOCK (XI	UNLOCK (XI	UNLOCK (XI

2.6.3.1 TEST-AND-SET INSTRUCTION

In most synchronization schemes, a physical entity must be used to represent the resource. This entity is often called a lock byte or semaphore. Thus, for each shared database or device there should be a separate lock byte. We will use the convention that lock byte = 0 means the resource is available, whereas lock byte = 1 means the resource is already in use. Before operating on such a shared resource, a process must perform the following actions :

1. Examine the value of the lock byte (either it is 0 or 1).
2. Set the lock byte to 1.
3. If the original value was 1, go back to step 1.

After the process has completed its use of the resource, it sets the lock byte to zero.

If we call the lock byte X, the action prior to use of the shared resource is called LOCK (X) and the action after use is UNLOCK (X). These are also called REQUEST (X) and RELEASE (X), respectively. These actions can be accomplished on the IBM System/370 by using the following instructions :

Note that it is essential that the lock byte not be changed by another process between locking steps 1 and 2. This requirement is met by the System/370 Test. and.

The lock byte is initially O. All possible timings of the processes should be considered

LOCK (X) :

1 Examine the value of the lock byte (either it is 0 or 1).

2 Set the lock byte to 1.

3 If the original value was 1, call WAIT (X). UNLOCK (X):

1 Set the lock byte to O.

2 Calls SIGNAL (X).

WAIT and SIGNAL are primitives of the traffic controller component of processor management. A WAIT (X) sets the process' PCB to the blocked state and links it to the lock byte X. Another process is then selected to run by the process scheduler.

2.6 **Check Your Progress.**

Fill in the Blanks.

1. Associated with processor allocation. and interprocess communication are two synchronization problems, ___________.
2. A ____________ occurs when the scheduling of two processes is so critical that the various orders of scheduling them result in different computations.

2.7 COMBINED JOB AND PROCESS SCHEDULING

In many instances, the job scheduling and process scheduling algorithms must interact closely.

If we attempt multiprogramming too many processes at the same time, especially in a demand paging system, all the processes will suffer from poor performance due to an overloading of available system resources (this phenomenon, called thrashing). On the other hand, in a conversational timesharing system, every user wants a "crack at the machine" as soon as possible-I-hour waits will not be tolerated.

As a compromise, we can adopt the philosophy that it is far better to give a few jobs at a time a chance than to starve them all! For example, we may allow two to five users to be ready at a time.

The basic concepts here are: (l) multiprogramming "in the small" for two to five jobs, process-scheduling every few ms; and (2) timeshare "in the large," where every few 100 ms or seconds one ready job is placed in hold status and one of the hold status jobs is made ready. (See Figure 2.13) Thus, over a period of a minute or so, every job has had its turn, and furthermore, the system runs fairly efficiently.

A SIGNAL (X) checks the blocked list associated with lock byte X; if there are any processes blocked waiting for X, one is selected and its PCB is set to the ready state. Eventually, the process scheduler will select this newly "awakened" process for execution.

The WAIT and SIGNAL mechanisms can be used for other purposes, such as waiting for I/O completion. After an I/O request (i.e., SIO instruction) has been issued, the process can be blocked by a WAIT (X) where X is a status byte associated with the I/O device (e.g., the lock byte can be stored in the device's Unit Control Block as shown in the sample operating system). When the I/O completion interrupt occurs, it is converted into a SIGNAL (X).

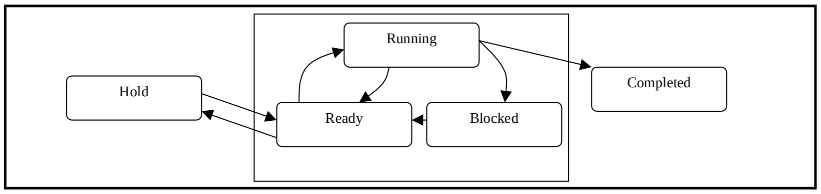


Figure 2.13 State Diagram

2.8 SUMMARY

Processor management has major functions: (1) job scheduling, (2) process scheduling, and (3) traffic control (such as interprocess communication and synchronization).

Job schedulers range from very simple, even manually done, algorithms to complex automated schedulers that consider the mix of jobs waiting to be run (i.e., in the hold state) as well as the available resources of the system. A uniprogramming operating system or simple priority-based process scheduler does not provide the same degree of optimization available in an adaptive process scheduling algorithm. In any type of system the interaction between job scheduling and process scheduling must always be carefully considered. If the job scheduler keeps dumping more and more jobs on an already overloaded process scheduler, system performance will suffer greatly.

Considerable research and study are still needed to resolve many of the problems connected with interprocess communication and synchronization. These features are very important to modern modular operating systems, as illustrated by their extensive usage in the sample operating system contained in Chapter

---

Source : *http://books.google.co.in (Google book)*
www.nou.edu.ng (ebook)

2.9 CHECK YOUR PROGRESS - ANSWERS

2.2

1. Traffic controller module
2. Memory, device, and processor management.

2.3

1. Microscopic scheduling

2. Roll out

2.4

1. Round robin
2. Inverse of remainder of quantum.

2.5

1. Device resources
2. Coordinated job scheduling

2.6

1. Race condition and deadly embrace.
2. Race condition

2.10 QUESTIONS FOR SELF - STUDY

1. Describe state model.
2. What is process scheduling?
3. Describe process synchronization in detail.
4. What's the difference between combined job and process scheduling?

2.11 SUGGESTED READINGS

1 . **Operating System Concepts** By Abraham Silberschatz, Peter B. Galvin & Greg Gagne.

2. **Operating systems** By Stuart E. Madnick, John J. Donovan

❑ ❑

Notes

Chapter : 3

Device Management

3.0 OBJECTIVES

In this chapter, we will discuss various types of device management techniques. After studying this chapter you will be able to -

- Analyse various types of device management techniques with device characteristics.
- Explain storage devices, channels and control units and device allocation.

3.1 INTRODUCTION

This chapter focuses on the management of I/0 devices. These include actual 1/0 devices, such as printers, card readers, tapes, disks, and drums, and supporting devices, such as control units or control channels. Some readers may already be familiar with some of these devices; our purpose in these sections is to merely give a feel for the relative speeds and characteristics of the important devices. A large installation may devote over half the system cost to I/0 devices. Therefore, it is desirable to use these devices in an efficient manner.

The basic functions of device management are :

1. Keeping track of the status of all devices, which requires special mechanisms. One commonly used mechanism is to have a database such as a Unit Control Block (UCB) associated with each device.
2. Deciding on policy to determine who gets a device, for how long, and when. A wide range of techniques is available for implementing these policies. For example, a policy of high device utilization attempts to match the nonuniform requests by processes to the uniform speed of many I/0 devices. There are three basic techniques for implementing the policies of device management:

A. Dedicated-a technique whereby a device is assigned to a single process.

B. Shared-a technique whereby a device is shared by many processes.

C. Virtual-a technique whereby one physical device is simulated on another physical device.

3. Allocation-physically assigning a device to process. Likewise the corresponding control units and channels must be assigned.
4. Deallocation policy and techniques. De allocation may be done on either a process or a job level. On a job level, a device is assigned for as long as the job exists in the system. On a process level, a device may be assigned for as long as the process needs it.

This chapter will focus mainly on the policies of device management, since the techniques are heavily device-dependent and are changing rapidly with hardware advances.

The module that keeps track of the status of devices 'is called the I/O traffic con-troller. We have called all modules associated with the operation of a single device the I/O device handlers. The I/O device handler's function is to create the channel program for performing the desired function, initiate i/0 to that device, handle the interrupts from it, and optimise its performance. In short, the device handler performs the physical 1/0. The I/O scheduler decides when an I/O processor is assigned to a request and sets up the path to the device.

3.2 TECHNIQUES FOR DEVICE MANAGEMENT

Three major techniques are used for managing and allocating devices: (I) dedicated, (2) shared, and (3) virtual.

3.2.1 Dedicated Devices

A dedicated device is allocated to a job for the job's entire duration. 30me devices lend themselves to this form of allocation. It is difficult, for example, to share a card reader, tape, or printer. If several users were to use the printer at the same time, would they cut up their appropriate output and paste it together? Unfortu-nately, dedicated assignment may be inefficient if the job does not fully and con-tinually utilize the device. The other techniques, shared and virtual, are usually preferred whenever they are applicable.

3.2.2 Shared Devices

Some devices such as disks, drums, and most other Direct Access Storage Devices (DASD) may be shared concurrently by several processes. Several processes can read from a single disk at essentially the same time. The management of a shared device can become quite complicated, particularly if utmost efficiency is desired. For example, if two processes simultaneously request a Read from Disk A, some mechanism must be employed. ` To determine which request should be handled first. This may be done partially by software (the I/O scheduler and traffic controller) or entirely by hardware (as in some computers with very sophisticated channels and control units).

Policy for establishing which process' request is to be satisfied first might be based on (1) a priority list or (2) the objective of achieving improved system out-put (for example, by choosing whichever request is nearest to the current position of the read heads of the disk).

3.2.3 Virtual Devices

Some devices that would normally have to be dedicated (e.g., card readers) may be converted into shared devices through techniques such as Spooling. For example, a Spooling program can read and copy all card input onto a disk at high speed. Later, when a process tries to read a card, the Spooling program intercepts the request and converts it to a read from the disk. Since several users may easily share a disk, we have converted a dedicated device to a shared device, changing one card reader into many "virtual" card readers. This technique is equally appli-cable to a large number of peripheral devices, such as teletypes, printers, and most dedicated slow input/output devices.

3.2.4 Generalized Strategies

Some professionals have attempted to generalize device management even more than is done here. For example, the call side of Spooling is similar to the file sys-tem and buffering bears striking similarity to Spooling. We could generalize even more and view memory, processors, and I/O devices as simple devices to which the same theory should apply. At present we realize that there are practical limitations to these generalizations, although at some time in the future more general theories may emerge that are applicable to all devices and that have direct practical applications.

3.2 Check Your Progress.

Fill in the blanks.

1. A _______ is allocated to a job for the job's entire duration.
2. Several processes may share some devices such as disks, drums, and most other __________ concurrently.
3. Some devices that would normally have to be dedicated (e.g., card readers) may be converted into shared devices through techniques such as _____________ .

3.3 DEVICE CHARACTERISTICS-HARDWARE CONSIDERATIONS

Almost everything imaginable can be, and probably has been, used as a peripheral device to a computer, ranging from steel mills to laser beams, from radar to mechan-ical potato pickers, from thermometers to space ships. Fortunately, most computer installations utilize only a relatively small set of peripheral devices. Peripheral devices can be generally categorized into two major groups: (1) input or output devices and (2) storage devices.

3.3.1 Input or Output Devices

An input device is one by which the computer "senses" or "feels" the outside world. These devices may be mechanisms such as thermometers or radar devices, but, more conventionally. They are devices to read punched cards, punched paper tape, or messages typed on typewriter like terminals. An output device is one by which the computer "affects" or "controls" the outside world. It may be a mechanism such as a temperature control knob or a radar direction control, but more commonly it is a device to punch holes in cards or paper tape, print letters and numbers on paper, or control the typing of typewriter like terminals.

3.3.2 Storage Devices

A storage device is a mechanism by which the computer may store information (a procedure commonly called writing) in such a way that this information may be retrieved at a later time (reading). Conceptually, storage devices are analogous to human storage mechanisms that use pieces of paper, pencils, and erasers.

It is helpful to differentiate among three types of devices. Our differen-tiation is based on the variation of access times (Tij) where:

Tij = time to access item j, given last access was to item i (or current position is item i)

We differentiate the following types of storage devices:

1. Serial Access' where Tij has a large variance (e.g., tape)

2. Completely Direct Access where Tij is constant (e.g., core)

3. Direct Access where Tij has only a small variance (e.g., drum)

3.3.2.1 Serial Access Device

A serial access storage device can be characterized as one that relies on strictly physical sequential positioning and accessing of information. Access to an arbitrary, stored item requires a "linear search" so that an average access takes the time required to read half the information stored. A Magnetic Tape Unit (MTU) is the most common example of a serial access storage device. The MTU is based upon the same principles as the audio tape deck or tape cassette, but instead of music or voices, binary information is stored. A typical serial access device is depicted in Figure 3.1.

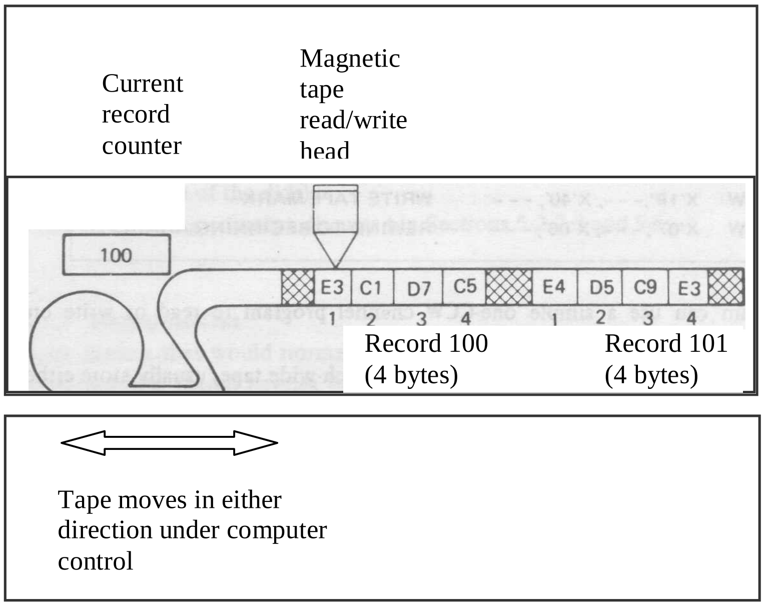


Figure 3.1- Magnetic tape units

Information is usually stored as groups of bytes, called records, rather than single bytes. In general, a record can be of arbitrary length, e.g., from 1 to 32,768 bytes long. In Figure 3.1 all the records are four bytes long. Records may be viewed as being analogous to spoken words on a standard audio tape deck. Each record can be identified by its physical position on the tape; the first record is 1, the second, 2, and so on. The current record counter of Figure 3.1 serves the same function as the "time" or "number-of feet" meter of audio tape decks. If the tape is positioned at it's beginning (i.e. completely rewound), and we wish to read record number 101, it is necessary to skip over all interviewing records (e.g., I, 2, . . .99, 100) in order to reach record number 101.

In addition to obvious I/O commands, such as read the Next Record or Write the Next Record, a tape unit usually has commands such as read the Last Record (read backward), Rewind to Beginning of Tape at High Speed, Skip Forward (or Backward) One Record without Actually Reading or Writing Data. In order to help establish groupings of records on a tape, there is a special record type called a tape mark. This record type can be written only by using the Write Tape Mark command. If the tape unit encounters a tape mark while reading or skipping, a special interrupt is generated. Tape marks are somewhat analogous to putting bookmarks in a book to help find your place.

3.3.2.2 Completely Direct Access Devices

A completely direct access device is one in which the access time Tij is a constant. Magnetic core memory, semiconductor memory, read-only wired memories, and diode matrix memories are all examples of completely direct access memories.

One property of a magnetic core is that when it changes magnetic state it induces a current would decompose the address:

Most contemporary computers use semiconductor (transistor like) memories. The basic ideas are similar to the core memories described above, but semi-conductor memories offer advantages in cost and speed. Typical semiconductor memories operate between 50 to 500 nanoseconds access time.

Although we say this type of storage device may be shared on the nanosecond level, it is limited in the sense that only one processor (CPU or I/O channel) can read a specific location (or block of locations) at a given time. Whenever more than one processor is accessing memory, especially if there is one CPU and several I/O channels, we frequently say that the channels are memory cycle stealing. Memory cycle stealing can degrade the performance of a system and is a factor with which the I/O traffic controller must contend.

3.3.2.3 Direct Access Storage Devoces (DASD)

A direct access device is one that is characterized by small variances in the access time Tij' these have been called Direct Access Storage Devices. Although they do not offer completely direct access, the name persists in the field.

In this section we give two examples of DASD, fixed head and moving-head devices.

3.3.2.3.1 Fixed-Head Drums and Disks

A magnetic drum can be simplistically viewed as several adjacent strips of mag-netic tape wrapped around a drum so that the ends of each tape strip join. Each tape strip, called a track, has a separate read/write head. The drum continuously revolves at high speed so that the records repeatedly pass under the read/write heads (e.g., record I, record 2, then record I again, record 2, . . .

etc.). A track number and then a record number identify each individual record. Typical magnetic drums spin very fast and have several hundred read/write heads; thus a random access to read or write can be accomplished in 5 or 10 ms in contrast to the minute required for random access to a record on a full magnetic tape.

3.3.2.3.2 Moving-head Disks and Drums

The magnetic disk is similar to the magnetic drum but is based upon the use of one or more flat disks, each with a series of concentric circles, one per read/write head. This is analogous to a phonograph disk. Since read/write heads are expensive, some devices do not have a unique head for each data track. Instead, the heads are phy-sically moved from track to track; such units are called moving-arm or moving-head DASDs. Each arm position is called a cylinder.

In order to identify a particular record stored on the moving-head DASD shown in Figure 3.4, it is necessary to specify the arm position, track number, and record number. Notice that arm position is based upon radial movement whereas record number is based upon circumferential movement. Thus, to access a record, it is necessary to move to the correct radial position (if not already there) and then to wait for the correct record to rotate under the read/write head.

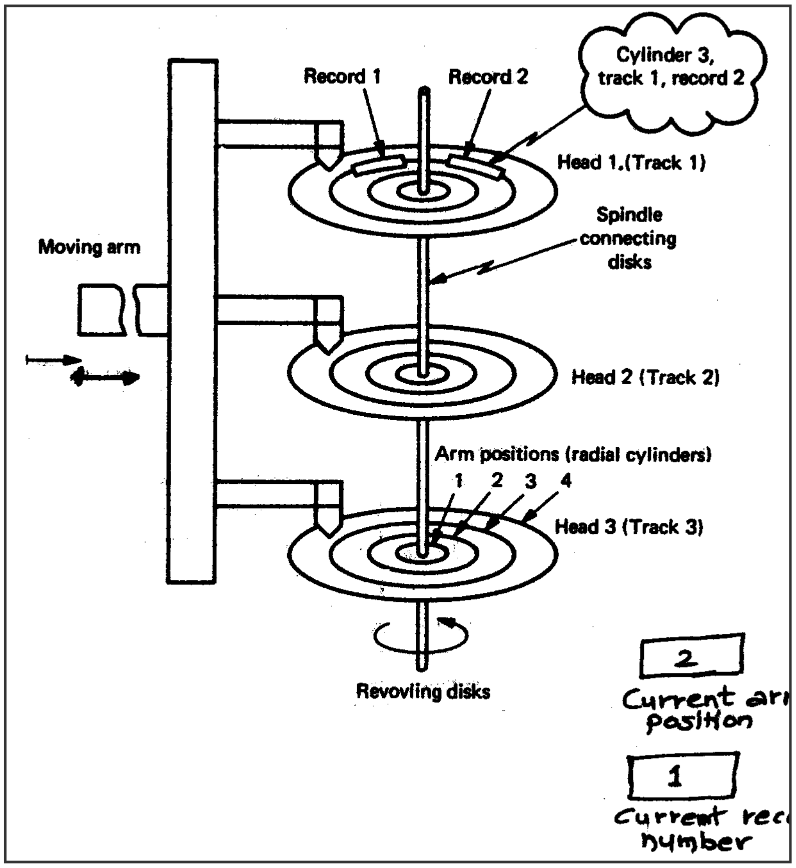


Figure 3.4 Moving-heed disk DASD

3.3.2.3.3 General Characteristics

Some DASD devices allow keys to be associated with each record. The keys are assigned by the user and can denote the contents of a record. The controller for such devices allows commands to read or write a record addressed not by a track, arm, or cylinder, but rather by its key.

There are often some constraints on DASDs that allow this form of addressing, e.g., the read head must already be positioned on the right cylinder.

Given the present trend toward reductions in the price of CPUs, it is probably better to keep a table in memory and not to use such searching on the disks, thus keeping the cost of the channels down.

The preceding examples are representative of storage devices, since in spite of numerous variations, all have similar characteristics.

3.3 Check Your Progress.

Fill in the Blanks.

1. Peripheral devices can be generally categorized into two major groups _____________ .
2. A _________ is a mechanism by which the computer may store information (a procedure commonly called writing).
3. _____________ storage devices are normally used for applications that require mainly sequential accessing.
4. A _____________ can be simplistically viewed as several adjacent strips of magnetic tape wrapped around a drum.

3.4 CHANNELS AND CONTROL UNITS

Device management must also take responsibility for the channels and control units. The channels are the special processors that execute the channel programs (i.e., the CCWs). Due to the high cost of channels, there are usually fewer channels than devices. The channels must, therefore, be switched from one device to another. Further cost savings can be obtained by taking advantage of the fact that relatively few I/O devices are in use at one time. Thus, it is wise to separate the device's electronics from its mechanical components. The electronics for several devices can be replaced by a single control unit, which can be switched from one device to another. Figure 3.6 illustrates the relationship between channels, control units, and devices. Typically, there are up to eight control units per channel and up to eight devices of the same type per control unit.

In order to initiate an I/O operation, a path is required between memory and the device. For example, to access drum J requires the use of channel A and drum con-trol unit D. If an I/O operation to drum G were in progress, it would not be possi-ble to initiate I/O on drum J because channel A would not be available (the I/O on drum G requires use of channel A and control unit C).

The device management routines must keep track of the status of (1) channels, (2) control units, and (3) devices. Since the channels and control units are usually scarce, they frequently create a serious bottleneck. Several hardware techniques can be used to lessen this problem: independent device operation, device buffering, multiple paths, and block multiplexing.

3.4.1 Independent Device Operation

In many cases there is sufficient capability in the device to complete an I/O operation without further assistance from the channel or control unit. A disk seek operation, which might require 50 ms, only needs the channel long enough to transmit the seek address from memory to the disk-10 or 20 ms. The seek can be completed by the device alone. Likewise, there are operations that may require both the device and the control unit but not the channel for completion.

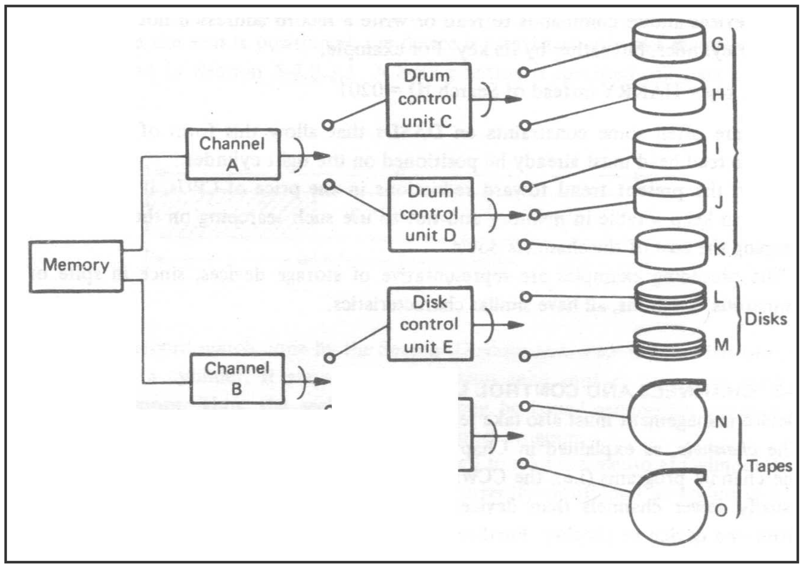


Figure 3.6 I/O configuration of a computer system

When an I/O operation is initiated, the entire path must be available. In order to indicate when parts of the path become available, there may be separate I/O com-pletion interrupts: channel end, control unit end, and device end. The status bits in the Channel Status Word designate the type of I/O completion interrupt. If more than one part of the path becomes available at the same time, there will be only one interrupt but the status bits will indicate all available parts (e.g., both the channel end and control unit end bits will be on in the CSW).

If a channel end occurs, the channel is no longer busy, although the device may still be busy. An I/O operation can then be initiated on some other device using that channel.

3.4.2 Buffering

By providing a data buffer in the device or control unit, devices that would ordinarily require the channel for a long period can be made to perform independently of the channel. For example, a card reader may physically require about 60 ms to read a card and transfer the data to memory through the channel.

However, a buffered card reader always reads one card before it is needed and saves the 80 bytes in a buffer in the card reader or control unit. When the channel requests that a card be read, the contents of the SO-byte buffer are transferred to memory at high speed (e.g., 100 / ms) and the channel is released. The device then proceeds to read the next card and buffer it independently. Card readers, cardpunches, and printers are frequently buffered.

3.4.3 Multiple Paths

We could reduce the channel and control unit bottlenecks by buying more channels and control units. A more economical alternative, however, is to use the channels and control units in a more flexible manner and allow multiple paths to each device. Figure 3.7 illustrates such an I/O configuration. In this example there are four different paths to Device E:

1. Channel A - Control Unit C - Device E
2. Channel A - Control Unit D - Device E
3. Channel B -Control Unit, C - Devices E
4. Channel B -Control Unit D - Devices E

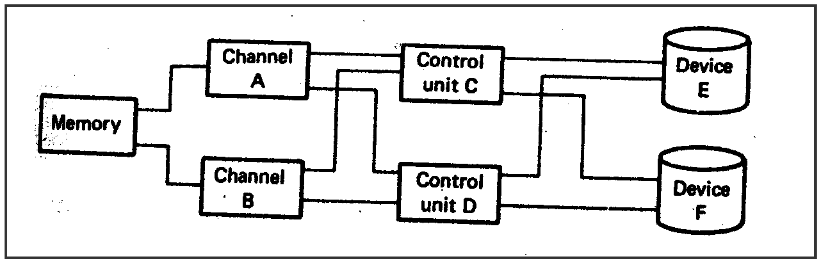


Figure 3.7 I/O configurations with multiple paths

If channel A and control unit C were both busy due to I/O for device F, a path could be found through channel B and control D to service device E. An actual I/O configuration may have eight control units per channel and eight devices per control unit rather than the sparse configuration. A completely flexi-ble configuration would make all control units accessible to all channels and all devices accessible to all control units, thus providing possibly 256 alternate paths between memory and each device. For economic reasons, this extreme case, though feasible, is seldom found.

In addition to providing flexibility, a multiple path I/O configuration is desirable for reliability. For example, if channel a malfunction and must be repaired, all the control units and devices can still be accessed through channel B. It is the responsi-bility of the device management routines to maintain status information on all paths in the I/O configuration and find an available' route to access the desired device.

3.4.4 Block Multiplexing

On conventional 370 selector channels, the techniques of independent device opera-tion and buffering are applicable only to the last CCW of a channel program. The channel and control unit remain connected to the device for the duration of the channel program. A 370 block multiplexor channel can be servicing multiple channel programs at the same time (e.g., eight channel programs). When the channel, encounters an I/O operation such as a buffered or independent device operation that does not need the channel for a while, it automatically switches to another channel pro-gram that needs immediate servicing. In essence, a block multiplexor channel repre-sents a hardware implementation of multiprogramming for channel programs.

This type of channel multiprogramming could be simulated on a selector channel by making all channel programs only one command long. The device management software routines could then reassign the channel as necessary. However, this approach is not very attractive unless the central processor is extremely fast, since the channel switching must be done very frequently and very quickly.

3.4 Check Your Progress.

Fill in the Blanks.

1. Typically, there are up to ____________ units per channel and up to eight devices of the same type per control unit. When an I/O operation is initiated, the entire path must be available.

2. We could reduce the channel and control unit bottlenecks by buying more ____________.

3.5 DEVICE ALLOCATION CONSIDERATIONS

As noted at the beginning of this chapter, there are three major techniques of device allocation: dedicated, Shared, or virtual.

Certain devices require either manual actions or time-consuming I/O operations to be performed prior to use by a job, such as manually placing a deck of cards in a card reader's hopper or positioning a magnetic tape to the desired data record. To switch use of such a device among several jobs would require an excessive amount of work. Thus, these devices are dedicated to one job at a time and are not reassigned until that job releases the device-usually at the termination of the job. The assignment of a device to a job is usually performed in conjunction with the job-scheduling routines of processor management.

Due to the relative ease and speed with which a direct access device such as a magnetic drum or disk can be repositioned, it is possible to Share it among several jobs running concurrently. A shared device is logically treated as if it were a group of dedicated devices. Each job using a shared device logically has a separate area to itself; the direct access property of the device makes it economical to switch back and forth among these areas. The device management routines may decide which shared device will be assigned to a job and what area of the device will be assigned. These decisions are also frequently made in conjunction with the job-scheduling routines.

For both dedicated and shared devices, static and dynamic allocation decisions must be made. The static decisions largely involve determining which dedicated and shared devices are to be assigned to the job. In general, device management tries to spread the allocation among devices so as to minimize channel and control unit bottlenecks. Often, the user can provide information on the job control cards indicating which of the devices will be used concurrently (these should be on separate channels) and which will be used at different times (these can be safely assigned to the same channel). The OS/360 Affinity (AFF) and Separation (SEP) job control language options serve this purpose.

Since channels and control units are so scarce, they are seldom permanently assigned to a job. Instead, an assignment is made when a job makes an I/O request to device management. The assigned channel and control unit are deallocated and available for reassignment as soon as the I/O request is completed.

The static allocation decisions of device management are analogous to the static decisions of the job scheduler in processor management. The dynamic allocation is similar in purpose to the process scheduler of processor management.

3.5 **Check Your Progress.**

Fill in the Blanks.

1. Certain devices require either _____________ actions or time-consuming __________ to be performed prior to use by a job,
2. The _____________ decisions largely involve determining which dedicated and shared devices are to be assigned to the job.
3. The static allocation decisions of device management are ___________ to the static decisions of the job scheduler in processor management.

3.6 I/O Traffic Controller, I/O Scheduler, I/O Device Handlers

The functions of device management can be conveniently divided into three parts :

1. I/O traffic controller-keeps track of the status of all devices, control units, and channels.
2. I/O scheduler-implements the policy algorithms used to allocate channel, control unit, and device access to I/O requests from jobs.
3. I/O devices handlers-perform the actual dynamic allocations, once the I/O scheduler has made the decision, by constructing the channel program, issuing the start I/O instruction, and processing the device interrupts. There is usually a separate device handler for each type of device.

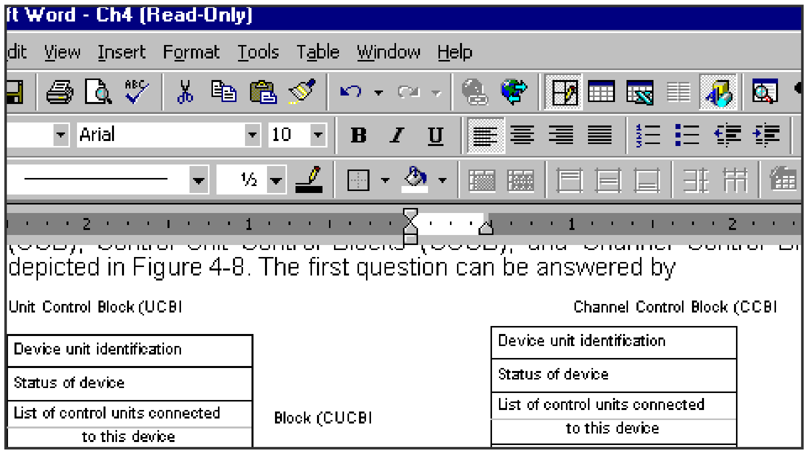
3.6.1 I/O Traffic Controller

Whereas the I/O scheduler is mainly concerned with policies (e.g., who gets the device and when), the I/O traffic controller is primarily concerned with mechan-ics (e.g., can the device be assigned?). The traffic controller maintains all status information.

The I/O traffic controller becomes complicated due to the interdependencies introduced by the scarcity of channels and control units, and the multiple paths The traffic controller attempts to answer at least three key questions :

1. Is there a path available to service an I/O request?
2. Is more than one path available?
3. If no path is currently available, when will one be free?

In order to answer these questions, the traffic controller maintains a database reflecting status and connections. This can be accomplished by means of Unit Control Blocks (UCB), Control Unit Control Blocks (CUCB), and Channel Control Blocks (CCB) as depicted in Figure 3.8. The first question can be answered by


ft Word - Ch4 (Read-Only)
dit View Insert Format Tools Table Window Help
Arial
10

depicted in Figure 4-8. The first question can be answered by

Unit Control Block (UCB)

Device unit identification
Status of device
List of control units connected to this device

Block (CUCB)

Channel Control Block (CCB)

Device unit identification
Status of device
List of control units connected to this device

Figure 3.8 Control blocks

starting at the desired UCB and working back through the connected control units and channels trying to find a combination that is available. In a similar manner it is possible to determine all available paths. This second question may be important, especially when the I/O configuration is not symmetric, as shown in Figure 3-9. Since choosing one path may block out other I/O requests. Thus, one path may conflict with fewer I/O requests than another path. In Figure 3.9, there may be up to five different paths to device H, yet only one possible path for either device G or I.

Under heavy I/O loads, it is quite likely that there will not be a path available at the time an I/O request is issued by a process. When an I/O completion interrupt occurs, one or more components (i.e., device, control units, and/or channel) become available again. We could try all I/O requests that are waiting to see if any can now be satisfied; if, however, hundred of I/O requests were waiting, this could be a time- consuming approach. A more efficient approach would be to list all I/O requests that are "interested" in the component in the control block, as suggested in Figure 3-8. Only those particular I/O requests would then have to be examined.

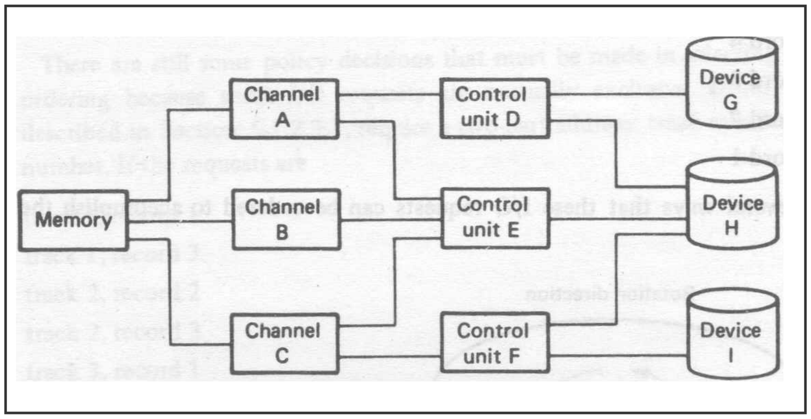


Figure 3.9 Asymmetric I/O configurations

3.6.2 I/0 Scheduler

If there are more I/O requests pending than available paths, it is necessary to choose which I/O requests to satisfy first. Many of the same concerns discussed in process scheduling are applicable here. One important difference is that I/O requests are not normally timesliced, that is, once a channel program 'has been started it is not usually interrupted until it has been completed. Most channel programs are quite' short and terminate within 50-100 ms; it is, however, possible to encounter channel programs that take seconds or even minutes before termination.

Many different policies may be incorporated into the I/0 scheduler. For example, if a process has a high priority assigned by the process scheduler, it is reasonable also to assign a high priority to its I/O requests. This would help complete that job as fast as possible.

Once the I/O scheduler has determined the relative orderings of the I/O requests, the I/O traffic controller must determine which, if any, of them can be satisfied. The I/O device handlers then provide another level of device-dependent I/O sched-uling and optimisation.

3.6.3 I/0 Device Handlers

In addition to setting up the channel command words, handling error conditions, and processing I/O interrupts, the I/O device handlers provide detailed scheduling algorithms that are dependent upon the peculiarities of the device type. There is usually a different device handler algorithm for each type of I/O device.

3.6.3.1 Rotational Ordering

Under heavy I/O loads, several I/O requests may be waiting for the same device. As noted earlier, there may be significant variations in the Tij access time for a device. Certain orderings of I/O requests may reduce the total time required to. service the I/O requests. Consider the drumlike device that has only four records stored in it (see Figure 3.10). The following four I/O requests have been received and a path to the device is now available :

1. Read record 4
2. Read record 3
3. Read record 2
4. Read record 1

There are several ways that these I/O requests can be ordered to accomplish the same result.

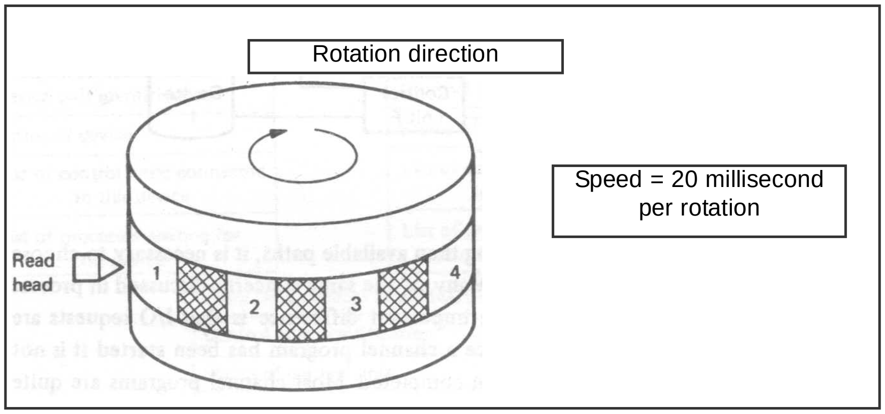


Figure 3.10 Simple drum like device illustrating rotational order

Ordering A

If the I/O scheduler satisfies the request by reading record 4, 3, 2, I, then the total processing time equals T?4 + T 43 + T 32 + T 21 "=" 3 revolutions (3/4 + 3 X 3/4 rotations), which equals approximately 60 ms. For the reading of record 4, since we do not know the current position of the drum, we assume 1/2 revolution (10 ms) to position it plus 1/4 revolution (5 ms) to read it.

Ordering B

If the I/O scheduler satisfies the request by reading record 1, 2, 3, 4, then the total processing time equals T1l + Tn + T23 + T34 "=" 1-1/2 revolutions (3/4 + 3 X 1/4 rotations), which equals approximately 30 ms.

The results of ordering A and ordering B are the same, but there is a differ-ence in speed of a factor of 2.

Ordering C

If we knew that "1" equalled record 3 (i.e., current read position is record 3), then the ordering 4, 1, 2, 3 (T34+T4+ T+ T23) would be even better exactly 1 revolution, which equals 20 ms.

In order to accomplish ordering C, it is necessary for the device handler to know the current position for the rotating drum. This hardware facility is called rotational position sensing.

Under heavy I/O loads, rotational ordering can increase throughput

There are still some policy decisions that must be made in selecting a rotational ordering because some I/O requests are mutually exclusive. Drum requests, require a two-part address: track number and record number. If the requests are

track 1, record 1
track 1, record 3
track 2, record 2
track 2, record 3
track 3, record I

then, requests 1 and 5 and requests 2 and 4 are mutually exclusive since they refer-ence the same record number. Thus, either request 1 can be serviced on the first rotation and request 5 on the second rotation, or vice versa. Sorting the requests on the basis of record number is often called slotting since there is one I/O request "slot" per record number per revolution. All I/O requests for the same record number position must "fight" for the same slot. This congestion can be decreased if the hardware allows reading and writing from more than one track at a time, but this capability usually requires the use of an additional channel and control unit as well as more electronics in the device.

Some devices, such as the IBM 2305 Fixed Head Disk, can perform some of the rotational ordering and slotting automatically in the hardware. The control unit for the 2305 can accept up to 16 channel programs at the same time, via a block multiplexor-channel, and it attempts to process these I/O requests in an "optimal" order.

3.6.3.2 Alternate Addresses

The access time to read an arbitrary record on a drumlike device is largely determined by the rotational speed. For a given device this speed is constant.

Recording each record at multiple locations on the device can substantially reduce the effective access time. Thus, there are several alternate addresses for reading the same data record. This technique has also been called folding.

Let us consider a device that has eight records per track and a rotation speed of 20 ms. If record A is stored on track 1, record 1, on the average it would take half a revolution-to ms-to access record A. If a copy of record A is stored at both track 1, record 1 and track 1, record 5, then by always accessing the "closest" copy, using rotational position sensing, the effective access time can be reduced to 5 ms. Similarly, .the effective access time can be reduced to 2.5 ms or even 1.25 ms by storing even more copies of the same data record.

Of course, by storing multiple copies of the same data record the effective capac-ity, in terms of unique data records, is reduced by the number of copies. If there are n copies per record, the device has been "folded" n times.

3.6.3.3 Seek Ordering

Moving-head storage devices have the problem of seek position in addition to rotational position. I/O requests require a three-part address: cylinder number, track number, and record number. If the requests

cylinder 1, track 2, record 1

cylinder 40, track 3, record 3

cylinder 5, track 6, record 5

cylinder 1, track 5, record 7

were processed in the order shown, a considerable amount of time would be spent to move the seek arm back and forth.

A more efficient ordering would be:

cylinder 1, track 2, record 1
cylinder 1, track 5, record 7
cylinder 5, track 6, record 5
cylinder 40, track 3, record 3

These requests are in Minimum seek time order, that is, the distances between seeks have been minimized.

There are many other possible seek orderings When rotational position is important, it is sometimes better to seek a further distance to get a record that will be at a closer rotational position. For an extreme example, if seek times were very small in comparison to the rotation time, the first ordering shown might be best since it accesses records in the order 1, 3, 5, 7. Determining this minimum service time order can be quite complex. It is further complicated by the fact that seek times, though monotonic, are not usually linear as a function of seek distance, i.e., the time to move the seek arm 20 cylinders is usually less than double the time to move it 10 cylinders.

In both the minimum seek time and minimum service time cases, the ordering was determined by the current I/0 requests for that device. New I/O requests are likely to occur while the original ones are being satisfied. In retrospect, the ordering chosen on the basis of the original I/O requests may be poor when the new I/0 requests are considered. Unless the system has a "crystal ball" feature, it is impossible to know what future requests will be. Several schemes have been used to minimize the negative effects of future I/0 requests.

Usually the ordering is revaluated after each individual request has been processed so as to allow consideration of any new requests. Unidirectional or linear sweep service time ordering is a technique that moves the arm monotonically in one direction. When the last cylinder is reached or there are no further requests in that direction, the arm is reset back to the beginning and another sweep is started. This technique attempts to decrease the amount of back and forward arm thrashing caused by the continued arrival of new I/0 requests.

3.6.4 Device Management Software Overheads

Each of the device management algorithms discussed in the preceding sections pro-vides additional device efficiency but at the cost of increased processor time (unless the algorithms are incorporated in the hardware, in which case there is increased hardware expense). The value of these algorithms depends upon the speed of the processor and the amount of I/O requests. If there are few I/O requests, multiprogramming of the processor alone may be sufficient to attain high throughput. If the processor is quite slow, the processor overhead may swamp the I/O advantages.

For example, a System/360 Model 30, with an average instruction time of 30 ms would take 30 ms to execute a 1000 instruction reordering algorithm. If the I/O requests take less than 30 ms to service without reordering, there is no savings attained. However, reordering techniques do become very important on large-scale computers.

Virtual storage memory management increases the load on device manage-ment in two ways. First, the page swapping often introduces a large amount of I/O requests. Second; the device handler must handle translation from virtual memory addresses to physical memory addresses in the construction of CCWs, since most channels require physical memory addresses rather than virtual memory addresses.

Finally, device management must ensure the protection of each job's areas in main memory and secondary storage. In the extreme, this would require very care-ful analysis of all I/O requests and addresses referenced. This can be done because all I/0 instructions are privileged and, thus, can be initiated only by the operating system. On the 360 it is possible to use the memory lock hardware to safeguard memory areas. Similar hardware features are often provided to protect secondary storage areas.

In total, device man-agement consumes a substantial amount of processor capacity. This overhead is usually worthwhile since device management can sig-nificantly increase system throughput by decreasing system I/O wait time.

3.6 **Check Your Progress.**

Fill in the Blanks.

1. _____________ keeps track of the status of all devices, control units, and channels.
2. I/O _____________ the policy algorithms used to allocate channel, control unit, and device access to I/O requests from jobs.

3.7 VIRUTAL DEVICES

3.7.1 Motivation

Devices such as card readers, punches, and printers present two serious problems that hamper effective device utilization.

First, each of these devices performs best when there is a steady, continuous stream of requests at a specified rate that matches. its physical characteristics (e.g., 1,000 cards per minute read rate, 1,000 lines per minute print rate, etc.). If a job attempts to generate requests faster than the device's performance rate, the job must wait a significant amount of time. On the other hand, if a job generates requests at a much lower rate, the device is idle much of the time and is substan-tially underutilized.

Second, these devices must be dedicated to a single job at a time. Thus, since most jobs perform some input and output, we would require as many card readers and printers as we have jobs being multiprogrammed. This would normally be uneconomical because most jobs use only a fraction of the device's capacity.

To summarize, these devices should have a steady request rate, whereas actual programs generate highly irregular requests. A small input/output-intensive program (e.g., a check-printing program) might generate thousands of print requests in a few seconds of CPU time. On the other hand, a compute-intensive program (e.g., de-termine the first 1,000 prime numbers) may print only a single line of output for every hour of CPU time. Buffering and multiprogramming, help reduce these problems, are not capable of solving them completely. Furthermore, many other devices, such as plotters, graphic displays, and type-writer terminals, have similar problems.

3.7.2 Historical Solutions

These problems would disappear, or at least substantially decrease, if it were possible to use Direct Access Storage Devices for all input and output. A single DASD can be efficiently shared and simultaneously used for reading and/or writing data by many jobs, as shown in Figure 3.11. Furthermore, DASDs provide very high performance rates, especially if the data are blocked, thereby decreasing the amount of wait time for jobs that require substantial amounts of input/output.

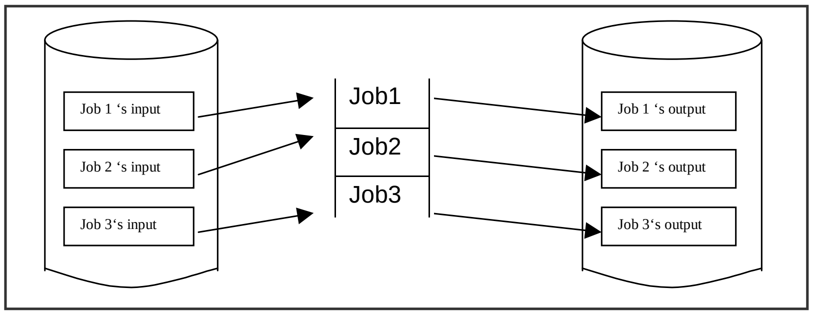


Figure 3.11 DASD Solutions

Although theoretically appealing, the use of DASDs for all input and output appears impractical. How do we get our input data onto the DASD in the first place? What do we do with the output data recorded on the output DASD? Consider for a moment walking into a drugstore with a disk pack and trying to convince the clerk that recorded on a specific part of the disk is your payroll check and you wish to cash it!

3.7.2.1 Offline Peripheral Operations

Fortunately, a solution to this dilemma can be found in the three-step process illustrated in Figure 3.12. At step 1 we use a separate computer whose sole function is to read cards at maximum speed and record the corresponding information on a DASD. Two or more card readers may be used by computer 1, depending upon the amount of input that must be transcribed.

At step 2 the DASD containing the input recorded by computer 1 is moved over to the main processing computer (computer 2). This step corresponds to the one shown in Figure 3-12. Multiple jobs can be in execution, each reading its respective input from the input DASD and writing its output onto an output DASD. Note in this example that it is possible to multiprogram three jobs even though only two physical card readers are assumed to exist (on computer 1 at step 1).

Finally, at step 3, the output DASD is moved to a third computer that reads the recorded output at high speed and prints the information on the printers.

The three-step process described above was used quite extensively during the 1960’s. The work performed by computers 1 and 3 was termed offline peripheral processing and the computers were called peripheral computers because they performed no computation but merely transferred information from one peripheral device to another. Since this operation was performed independently of the main computer, this was an "offline" operation.

There are several observations that can be made at this time. Inasmuch as the peripheral computers are required to perform only rather simple tasks, they can be quite simple, slow, and inexpensive. Furthermore, here need not always be two peripheral computers. If there is a relatively low peripheral load, one peripheral computer might handle it all-possibly switching from input processing to output processing every few hours. On the other hand, if substantial peripheral processing is required, many peripheral computers may be used.

During the early 1960’s, a typical installation, such as that of MIT, might have consisted of three peripheral computers (e.g., IBM 140’s) to service one main computer (e.g., IBM 7094). At that time magnetic tapes were used instead of DASDs, although the basic concepts were still very similar to the approach described above.

The offline peripheral processing technique solved the problems presented earlier, but it also introduced several new problems in regard to (1) human intervention, (2) turnaround, and (3) scheduling. Since human operators were required to move the input DASDs from the input peripheral computer to the main com-puter and to perform a simpler task for output processing, there were many op-portunities for human error and inefficiency. Many jobs i.e., a batch of jobs) were recorded on the input DASD before it was moved to the main computer. The entire input DASD had to be processed and the output DASD filled before the latter was moved to the output peripheral computer. Then, finally, the individual job's output could be separated and given to its owner.

This batch processing approach, though efficient from the computer's point of view, made each job wait in line at each step and often increased its turnaround time. As a result of this hatching processing, it was difficult to provide the desired priority scheduling. e.g., if two high-priority jobs were to be run but were in separate batches, one would have to wait until the other's batch was completely processed.

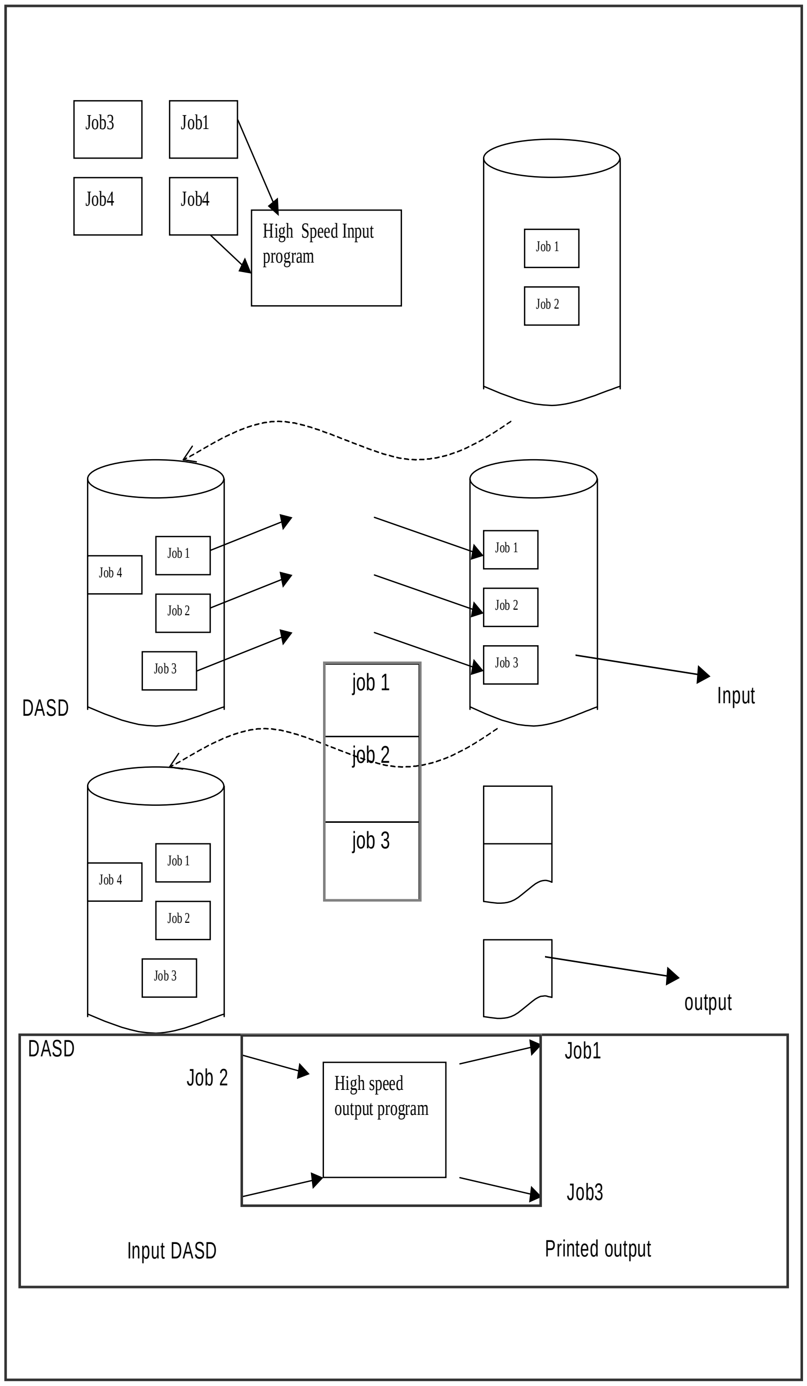


Figure 3.12 Offline Peripheral Operations Solution

3.7 **Check Your Progress.**

Fill in the blanks.

1. These devices should have a steady _____________, whereas actual programs generate highly irregular requests
2. There need not always be ______ peripheral computers.
3. The offline peripheral processing technique solved the problems presented earlier, but it also introduced several new problems in regard to _____________.

3.8 SUMMARY

This chapter has described techniques for handling the four basic functions of device management. We presented a framework for handling these functions, based upon three software modules :

1. I/0 traffic controller
2. I/0 scheduler
3. I/0 device handler

The reader should not assume that this is the only way or the best way to handle these functions. However, we feel that this framework is conceptually the most desirable, and the one that many contemporary systems follow.

3.9 CHECK YOUR PROGRESS - ANSWERS

3.2

1. Dedicated device
2. Direct Access Storage Devices (DASD)
3. SPOOLing

3.3

1. (1) input or output devices and (2) storage devices.
2. Storage device
3. Serial access
4. Magnetic drum

3.4

1. Eight con-trol units
2. Channels and control units.

3.5

1. Manual actions or I/O operations
2. Static
3. Analogous

3.6

1. I/O traffic controller
2. I/O scheduler-implements

3.7

1. Request rate
2. Two
3. (1) human intervention, (2) turnaround, and (3) scheduling.

3.10 QUESTIONS FOR SELF - STUDY

1. Discuss virtual and dedicated devices.
2. What you mean by direct access storage?
3. Discuss moving head disks and drums.
4. What are multiple paths?
5. What is virtual system?

3.11 SUGGESTED READINGS

1. **Operating System Concepts** By Abraham Silberschatz, Peter B. Galvin & Greg Gagne.
2. **Operating systems** By Stuart E. Madnick, John J. Donovan

❑ ❑ ❑

Notes

Notes

Chapter : 4

Memory Management

4.0 OBJECTIVES

After studying this chapter you will be able to -

- Discuss the concept of memory management
- Explain the concept of paging, swapping, segmentation
- State the importance of virtual memory and page replacement algorithms.

4.1 INTRODUCTION

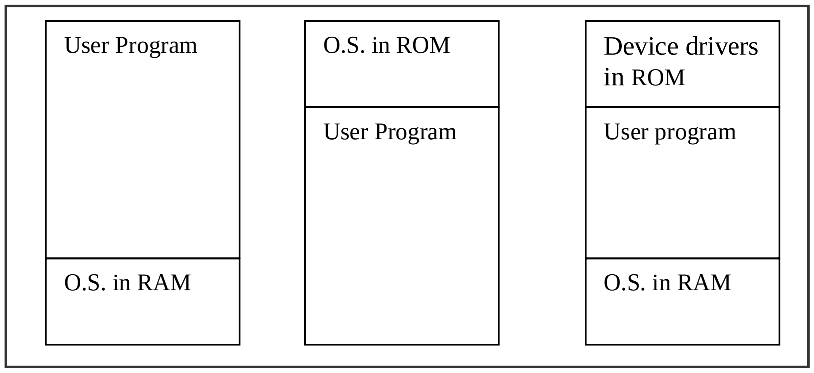
In this chapter we will investigate a number of different memory management schemes, ranging from very simple to highly sophisticated. We will start at the beginning and look first at the simplest possible memory management system and then gradually progress to more and more elaborate ones.

4.2 BASIC MEMORY MANAGEMENT

Memory management systems can be divided into two classes: those that move processes back and forth between main memory and disk during execution (swapping and paging), and those that do not. The latter are simpler, so we will study them first. Later in the chapter we will examine swapping and paging. Throughout this chapter the reader should keep in mind that swapping and paging are largely artifacts caused by the lack of sufficient main memory to hold all the programs at once. As main memory gets cheaper, the arguments in favor of one kind of memory management scheme or another may become obsolete-unless programs get bigger faster man memory gets cheaper.

4.2.1 Monoprogramming Without Swapping or Paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Figure. 4-1 (a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Figure. 4-1(b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down elow. The lat-ter model is used by small MS-DOS systems, for example. On IBM PCs, the por-tion of the system in the ROM is called the **BIOS** {Basic Input Output System).



(a) **(b)** **(c)**

Figure 4.1

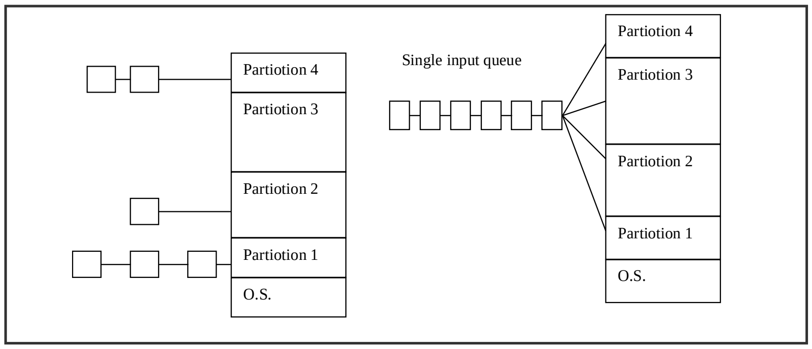
When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process fin-ishes, the operating system displays a prompt character and waits for a new com-mand. When it receives the command, it loads a new program into memory, over-writing the first one.

4.2.2 Multiprogramming with Fixed Partitions

Although monoprogramming is sometimes used on small computers with simple operating systems, often it is desirable to allow multiple processes to run at once. On timesharing systems, having multiple processes in memory at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU. Thus multiprogramming increases the CPU utilization. However, even on personal computers it is often useful to be able to run two or more programs at once.

This easiest way to achieve multiprogramming is simply to divide memory up into n (possibly unequal) partitions. This partitioning can, for example, be done manually when the system is started up.

When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partitions are fixed in this scheme, any space in a partition not used by a job is lost. In Figure. 4.2{a) we see how this system of fixed partitions and separate input queues looks.



(a) **(b)**

Figure 4.2

(a) Fixed memory partitions with separate input queues for each partition,

(b) Fixed memory partitions with a single input queue

The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3 in Figure. 4-2(a). An alternative organization is to maintain a single queue as in Figure. 4-2(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. Since it is undesirable to waste a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits. Note that the latter algorithm discriminates against small jobs as being unworthy of having a whole partition, whereas usually it is desirable to give the smallest jobs (assumed to be interactive jobs) the best service, not the worst.

One way out is to have at least one small partition around. Such a partition will allow small jobs to run without having to allocate a large partition for them. Another approach is to have a rule slating that a job that is eligible to run may not be skipped over more than k times. Each time it is skipped over, it gets one point.

When it has acquired k points, it may not be skipped again.

This system, with fixed partitions set up by the operator in the morning and not changed thereafter, was used by OS/360 on large IBM mainframes for many years. It was called MFT (Multiprogramming with a fixed number of Tasks or OS/MFT). It is simple to understand and equally simple to implement: incoming jobs are queued until a suitable partition is available, at which time the job is loaded into that partition and run until it terminates: Nowadays, few, if any operating systems, support this model.

Relocation and Protection :

Multiprogramming introduces two essential problems that must be solved relocation and protection.

For example, suppose that the first instruction is a call to a procedure at abso-lute address 100 within the binary file produced by the linker. If this program is loaded in partition 1, that instruction will jump to absolute address 100, which is inside the operating system. What is needed is a call to 100K + 100. If the program is loaded into partition 2, it must be carried out as a call to 200K + 100, and so on. This problem is known as the relocation problem.

One possible solution is to actually modify the instructions as the program is loaded into memory. Programs loaded into partition 1 have 100K added to each address; programs loaded into partition 2 have 200K added to addresses, and so forth. To perform relocation during loading like this, the linker must include in the binary program a list or bit map telling which program words are addresses to be relocated and which are opcodes, constants, or other items that must not be relocated. OS/MFT worked this way. Some microcomputers also work like this.

Relocation during loading does not solve the protection problem. A malicious program can always construct a new instruction and jump to it. Because programs in this system use absolute memory addresses rather than addresses related to a register, there is no way to stop a program from building an instruction that reads or writes any word in memory.

The solution, that IBM chose for protecting the 360 was to divide memory into blocks of 2K bytes and assign a 4-bit protection code to each block. The PSW contained a 4-bit key. The 360 hardware trapped any attempt by a running proc-ess to access memory whose protection code differed from the PSW key. Since only the operating system could change the protection codes and key, user proc-esses were prevented from interfering with one another and with the operating system itself.

An alternative solution to both the relocation and protection problems is to equip the machine with two special hardware registers, called the base and limit registers. When a process is scheduled, the base register is loaded with the ad-dress of the start of its partition, and the limit register is loaded with the length of the partition. Every memory address generated automatically has the base register contents added to it before being sent to memory. Thus if the base register is 100K, a CALL 100 instruction is effectively turned into a CALL 100K + 100 in-struction, without the instruction itself being modified. Addresses are also checked against the limit register to make sure that they do not attempt to address memory outside the current partition. The hardware protects the base and limit registers to prevent user programs from modifying them.

The CDC 6600-the world's first supercomputer-used this scheme. The Intel 8088 CPU used for the original IBM PC used a weaker version of this scheme-base registers, but no limit registers. Starting with the 286, a better scheme was adopted.

With a batch system, organizing memory into fixed partitions is simple and effective. Each job is loaded into a partition when it gets to the head of the queue. It stays in memory until it has finished. As long as enough jobs can be kept in memory to keep the CPU busy all the time, there is no reason to use anything more complicated.

4.2 Check Your Progress.

Fill in the blanks.

1. The operating system may be at the bottom of memory in ___________.
2. This easiest way to achieve multiprogramming is simply to divide memory up into n (possibly unequal) _______
3. Such a partition will allow ___________ to run without having to allocate a large partition for them.

4.3 SWAPPING

With timesharing systems or graphically oriented personal computers, situation is different. Sometimes there is not enough main memory to hold all the ' currently active processes, so excess processes must be kept on disk and brought in to run dynamically.

Two general approaches to memory management can be used, depending (in part) on the available hardware. The simplest strategy, called swapping, consists bringing in each process in its entirety, running it for a while, and then putting it back on the disk. The other strategy, called virtual memory, allows programs to run even when they are only partially in main memory.

The operation of a swapping system is illustrated in Fig. 4-3. Initially only process A is in memory. Then processes B and C are created or swapped in from disk. In Figure. 4-3(d) A terminates or is swapped out to disk. Then D comes in and B goes out. Finally E is the main difference between the fixed partitions of Figure. 4-2 and the variable partitions of Figure. 4-3 is that the number, location, and size of the partitions vary dynamically in the latter as processes come, and go, whereas they are fixed in the former. The flexibility of not being tied to a fixed number of partitions that may be too large or too small improves memory utilization, but it also complicates allocating and deallocating memory.

When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible.

This technique is known as memory compaction. It is usually not done because it requires a lot of CPU time A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple: you allocate exactly what is needed, no more and no less. If, however, processes" data segments can grow, for example, by dynamically allocating memory from a heap, as in many programming languages, a problem occurs whenever a process tries to grow. If a hole is adjacent to the process, it can be allocated and-the process allowed growing into the hole.

If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory. However, when swapping processes to disk, only the memory actually in use should be swapped; ft is wasteful to swap the extra memory as well. In Fig. 4-4(a) we see a memory configuration in which-space for growth has been allocated to two processes.

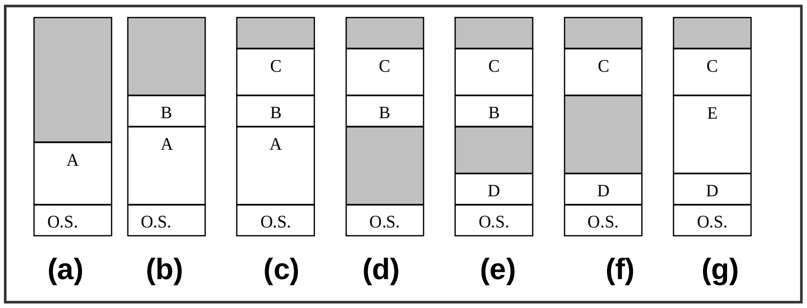


Fig. 4.3 The operation of a swapping system

If processes can have two growing segments, for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses, an alternative arrangement suggests itself, namely that of Figure. 4-4{b). In this figure we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing upward. The memory between them can be used for either segment. If it runs out, either the process will have to be moved to a hole with enough space, swapped out of memory until a large enough hole can be created, or killed.

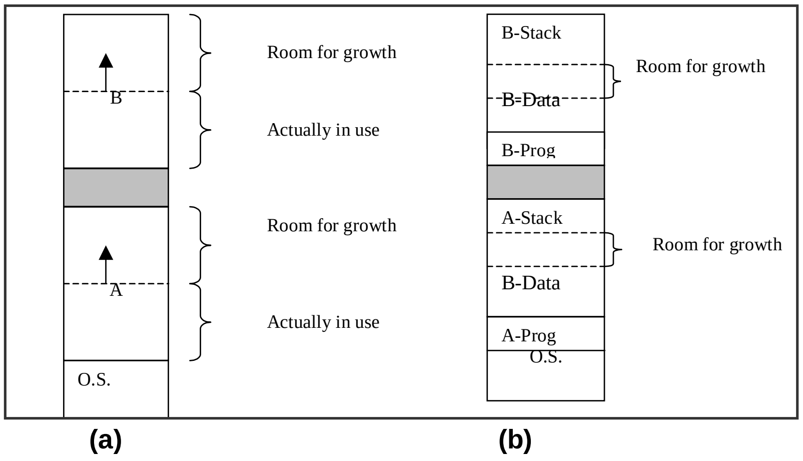


Fig. 4.4 (a) & (b) Allocating space for a growing data segment

4.3.1 Memory Management with Bit Maps

When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: bit maps and free lists. In this section and the next one we will look at these two methods in turn.

With a bit map, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bit map, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 4.5 shows part of memory and the corresponding.

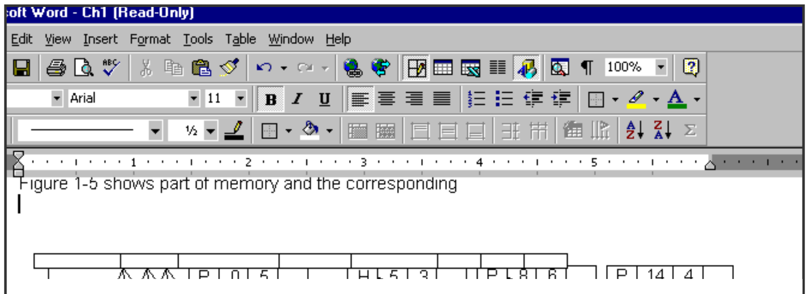

soft Word - Ch1 (Read-Only)
Edit View Insert Format Tools Table Window Help
100%
Arial
11
P 0 5
H 5 3
P 8 6
P 14 4

Figure 1-5 shows part of memory and the corresponding

Fig. 4.5 Memory Management with Bit Maps

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bit maps. However, even with an allocation unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map. A memory of 32n bits will use n map bits, so the bit map will take up only 1/33 of memory. If the allocation unit is chosen large, the bit map will be smaller, but appreciable memory may be wasted in the last unit if the process size is not an exact multiple of the allocation unit.

A bit map provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bit map depends only on the size of memory and the size of the allocation unit. The main problem with it is that when it has been decided to bring a k unit process into memory, the memory manager must search the bit map to find a run of k consecutive 0 bits in the map. Searching a bit map for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bit maps.

4.3.2 Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. Each entry in the list specifies a hole (H) or process (P), file address at which it starts, the length, and a pointer to the next entry.

In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. Three entries are merged and two items are removed from the list. Since the process table slot for the terminating process will normally point to the list entry for the process itself, it may be more convenient to have the list as a double-linked list, rather than the single-linked list. This structure makes it easier to find the previous entry and to see if a merge is possible.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created or swapped in process. We assume that the memory manager knows how much memory to allocate. The simplest algorithm is first fit. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

A minor variation of first fit is next fit. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does. Simulations by Bays show that next fit gives slightly worse performance than first fit.

Another well-known algorithm is best fit. Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Best fit is slower than first fit because it must search the entire list every time it is called. Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes. First fit generates larger holes on the average.

To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about worst fit, that is, always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

Maintaining separate lists for proc-esses and holes can speed up all four algorithms. In this way, all of them devote their full energy to inspecting holes, not processes. The inevitable price that is paid for this speedup on allocation is the additional complexity and slow down when deallocating memory, since a freed segment has to be removed from the process list and inserted into the hole list.

If distinct lists are maintained for processes and holes, the hole list may be kept sorted on size, to make best fit faster. When best fit searches a list of holes from smallest to largest, as soon as it finds a hole that fits, it knows that the hole is the smallest one that will do the job, hence the best fit. No further searching is needed, as it is with the single list scheme. With a hole list sorted by size, first fit and best fit are equally fast, and next fit is pointless.

When the holes are kept on separate lists from the processes, a small optimi-zation is possible. Instead of having a separate set of data structures for maintain-ing the hole list, the holes themselves can be used. The first word of each hole could be the hole size, and the second word a pointer to the following entry. The nodes of the list require three words and one bit (P/H), are no longer needed.

Yet another allocation algorithm is quick fit, which maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4K holes, the second entry is a pointer to a list of 8K holes, the third entry a pointer to 12K holes, and so on. Holes of say, 21K, could either be put on the 20K list or on a special list of odd-sized holes. With quick fit, finding a hole of. the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

4.3 Check Your Progress.

Fill in the blanks.

1. ___________ Consists bringing in each process in its entirety, running it for a while, then putting it back on the disk.
2. When swapping creates multiple __________ in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible.
3. If a process can not grow in memory and the ___________ on the disk is full, the process will have to wait or be killed.

4.4 VIRTUAL MEMORY

Many years ago people were first confronted with programs that were too big to fit in the available memory. The solution usually adopted was to split the program into pieces, called overlays. Overlay 0 would start running first. When it was done, it would call another overlay. Some overlay systems were highly complex, allowing multiple overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the operating system, dynamically, as needed.

Although the actual work of swapping overlays in and out was done by the system, the work of splitting the program into pieces had to be done by the programmer. Splitting up large programs into small, modular pieces was time consuming and boring. It did not take long before someone thought of a way to turn the whole job over to the computer.

The method that was devised has come to be known as virtual memory. The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. For example, a 16M program can run on a 4M machine by carefully choosing which 4M to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

Virtual memory can also work in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for part of itself to be brought in, it is waiting for I/O and cannot run, so the CPU can be given to another process, the same way as for any other multiprogramming system.

4.4.1 Paging

Most virtual memory systems use a technique called paging, which we will now describe. On any computer, there exists a set of memory addresses that programs can produce. When a program uses an instruction like
MOVE REG, 1000

it is copying the contents of memory address 1000 to REG (or vice versa, depending on the computer). Addresses Can be generated using indexing, base registers, segment register and other ways.

These program-generated addresses are called virtual addresses and form the virtual address space. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written.

A very simple example of how this mapping works is shown in Figure. 4-7. In this example, we have a computer that can generate 16-bit addresses, from 0 up to 64K. These are the virtual addresses. This computer, however, has only 32K of physical memory, so although 64K programs can be written, they cannot be loaded into memory in their entirety and run.

The virtual address space is divided up into units called pages. The corresponding units in the physical memory are called page frames. The pages and page frames are always exactly the same size. In this example they are 4K, but page sizes from 512 bytes to 64K are commonly used in existing systems. With 64K of virtual address space and 32K of physical memory, we have 16 vir-tual pages and 8 page frames. Transfers between memory and disk are always in units of a page. When the program tries to access address 0, for example, using the instruction.

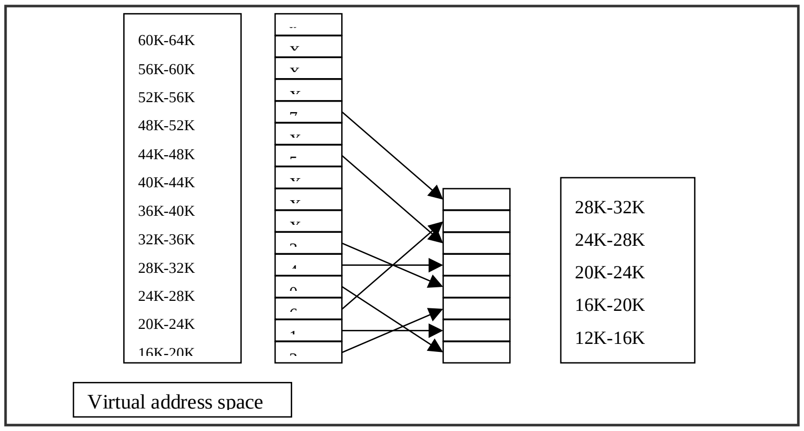


Fig. 4.7

MOVE REG, 0

The virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory board knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

4.7 MEMORY MAPPING

Similarly, an instruction

MOVE REG, 8192

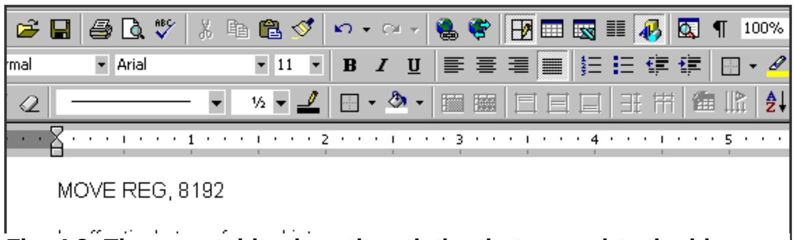
Is effectively transformed into

MOVE REG, 24576

Because virtual address 8192 is in virtual page 2 and this page is mapped onto physical page frame 6 (physical addresses 24576 to 28671). Another example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

By itself, this ability to map the 16 virtual pages onto any of the eight page frames by setting the MMU's map-appropriately does not solve the problem that the virtual address space is larger than the physical memory. Since we have only eight physical page frames, only eight of the virtual pages in Figure 4-8 are mapped onto physical memory. The others, shown as a cross in the figure 4-8, are not mapped. In the actual hardware, a Present/absent bit in each entry keeps track of whether the page is mapped or not. What happens if the program tries to use an unmapped page, for example, by using the instruction

MOVE REG.32780



MOVE REG, 8192

Fig. 4.8 The page table gives the relation between virtual addresses and physicalmemory addresses

Which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped (indicated by a cross in the figure 4-8), and causes the CPU to trap to the operating system. This trap is called a page fault. The operating system picks a little-used page frame and writes its contents back to the disk. It then fetches the page just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

Now let us look inside the MMU to see how it works and why we have chosen to use a page size that is a power of 2. In Figure. 4.9 we see an example of a virtual-address, 8196 (0010000000000100 in binary), being mapped using the MMU map of Figure. 4.8 .

4.4.2 Page Tables

In theory, the mapping of virtual addresses onto physical addresses is as we have just described it. The virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits). The virtual page number is used as an index into the page table to find the entry for that virtual page.

The purpose of the page table is to map virtual pages onto page frames.

Mathematically speaking, the page table is a function, with the virtual page num-ber as argument and the physical frame number as result. Using the result of this function, the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address. Despite this simple description, two major issues must be faced:

1. The page table can be extremely large.
2. The mapping must be fast.

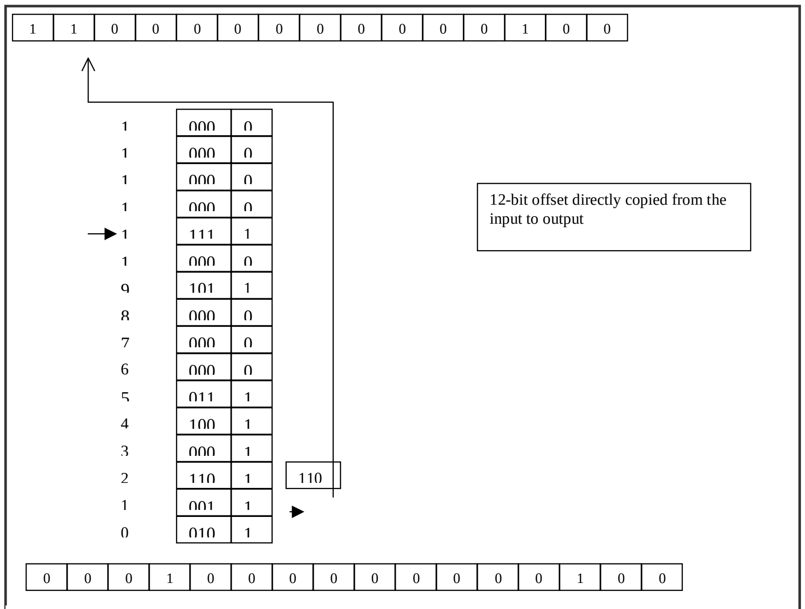


Fig. 4-9 The internal operation of the MMU with 16, 4K pages

The first point follows from the fact that modern computers use virtual addresses of at least 32 bits. With, say, a 4K-page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than you want to contemplate. With 1 million pages in the virtual address space, the page table must have 1 million entries. And remember that each process needs its own page table.

The second point is a consequence of the fact that the virtual-to-physical mapping must be done on every memory reference. A typical instruction has an in-struction word, and often a memory operand as well. Consequently, it is necessary to make 1, 2, or sometimes more page table references per instruction. If an instruction takes, say, 10 nanosecond, the page table lookup must be done in a few nano-seconds to avoid becoming a major bottleneck.

The need for large, fast page mapping is a significant constraint on the way computers are built. Although the problem is most serious with top-of-the-line machines, it is also an issue at the low end as well, where cost and price/perfor-mance are critical. In this section and the following ones, we will look at page table design in detail and show a number of hardware solutions that have been used in actual computers.

The simplest design (at least conceptually) is to have a single page table consisting of an array of fast hardware registers, with one entry for each virtual page, indexed by virtual page number. When a process is started up, the operating system loads the registers with the process's page table, taken from a copy kept in main memory. During process execution, no more memory references are needed for the page table. The advantages of this method are that it is straightforward and requires no memory references during mapping. A disadvantage is that it is potentially expensive (if the page table is large). Having to load the page table at every context switch can also hurt performance.

At the other extreme, the page table can be entirely in main memory. All the hardware needs then is a single register that points to the start of the page table. This design allows the memory map to be changed at a context switch by reloading one register. Of course, it has the disadvantage of requiring one or more memory references to read page table entries during the execution of each instruction. For this reason, this approach is rarely used in its most pure form, but below we will study some variations that have much better performance.

Multilevel Page Tables

To get around the problem of having huge page tables in memory all the time, many computers use a multilevel page table. A simple example is shown in Figure. 4.10. In Figure. 4.10 we have a 32-bit virtual address that is partitioned into a 10-bit PT1 field, a 10-bit PT2 field, and a 12-bit Offset field. Since offsets are 12 bits, pages are 4K, and there are a total of 220 of them.

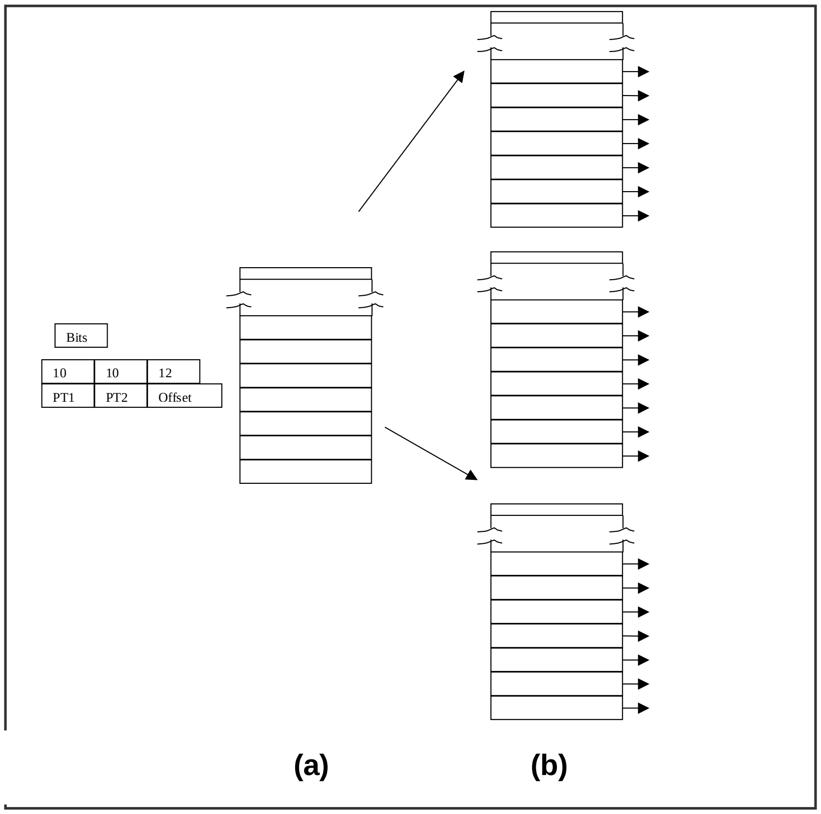


Fig. 4.10 - (a) A 32-bit address with two page table fields, (b) Two-level page tables

The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. Suppose, for example, that a process needs 12 megabytes, the bottom 4 megabytes of memory for program text, the next 4 megabytes for data and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

In Figure. 4.10(b) we see how the two-level page table works in this example. On the left we have the top-level page table, with 1024 entries, corresponding to the 10-bit PTl Field. When a virtual address is presented to the MMU, it first extracts the PTl field and uses this value as an index into the top-level page table. Each of these 1024 entries represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 1024 bytes. The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. Entry 0 of the top-level page table points to the page table for the program text, entry 1 points to the page table for the data, and entry 1023 points to the page table for the stack. The other entries are not used. The PT2 field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

The interesting thing to note about Figure 4.10 is that although the address space-contains over a million pages, only four page tables are actually needed: the top-level table, and the second-level tables for 0 to 4M, 4M to 8M, and the top 4M. The Present/absent bits in 1021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed. Should this occur, the operating system will notice that the process is trying to reference memory that it is not sup-posed to and will take appropriate action, such as sending it a signal or killing it. In this example we have chosen round numbers for the various sizes and have picked PTI equal to PT2 but in actual practice other values are also possible, of course.

The two-level page table system of Figure 4.10 can be expanded to three, four, or more levels. Additional levels give more flexibility, but it is doubtful that the additional complexity is worth it beyond three levels.

Let us now turn from the structure of the page tables in the large, to the details of a single page table entry. The exact layout of an entry is highly machine dependent, but the kind of information present is roughly the same from machine to machine. We give a sample page table entry. The size varies from computer to computer, but 32 bits is a common size. The most important field is the Page frame number. After all, the goal of the page mapping is to locate this value. Next to it we have the Present/absent bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not cur-rently in memory. Accessing a page table entry with this bit set to 0 causes a page fault. The Protection bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement r having 3 bits, one bit each for enabling reading, writ-ing, and executing the page.

The *Modified* and Referenced bits keep track of page usage. When a page is written to, the hardware automatically sets the Modified bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified (i.e., is "dirty"), it must be written back to the disk. If it has not been modified (i.e., is "clean"), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the dirty bit, since it reflects the page's state.

The Referenced bit is set whenever a page is referenced, either for reading or writing. Its value is to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms that we will study later in this chapter.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached object. With this bit, caching can be turned off. Machines that have a separate I/O space and do not use memory mapped I/O do not need this bit.

Note that the disk address used to hold the page when it is not in memory is not part of the page table. The reason is simple. The page table holds only that information the hardware needs to translate a virtual address to a-physical-address. Information the operating system needs to handle page faults is kept in software tables inside the operating system.

4.4.3 TLBs-Translation Look aside Buffers

In most paging schemes, the page tables are kept in memory, due to their large size. Potentially, this design has an enormous impact on performance. Consider, for example, an instruction that copies one register to another. In the absence of paging, this instruction makes only one memory reference, to fetch the instruction. With paging, additional memory references will be needed to access the page table. Since execution speed is generally limited by the rate the CPU can get instructions and data out of the memory, having to make two page table references per memory reference reduces performance by 2/3, Under these conditions, no one would use it.

Computer designers have known about this problem for years and have come up with a solution. Their solution is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around.Thus only a small fraction of the page table entries are heavily read, the rest is barely used at all.

The solution that has been devised is to equip computers with a small hard-ware device for mapping virtual addresses to physical addresses without going through the page table. The device, called a TLB (Translation Look aside Buffer) or sometimes an associative memory, is illustrated in Figure. 4.11. It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 64. Each entry contains information about one page, in particular, the virtual page number, a bit that is set when the page is mod-ified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located. These Fields have a one-to-one correspondence with the fields in the page table. Another bit indicates whether the entry is valid (i.e., in use) or no.

Valid	Virtual page	Modified	Protectio n	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 4.11 A TLB lo speed up paging

An example that might generate the TLB of Figure. 4.11 is a process in a loop that spans virtual pages 19, 20, and 21, so these TLB entries have protection codes for reading and executing. The main data currently being used (say, an array being processed) are on further pages.

Let us now see how the TLB functions. When a virtual address is presented to the MMU for translation, the hardware first checks to see if it's virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel).. If a valid match is found and the access does not violate the pro-tection bits, the page frame is taken directly from the TLB, without going to the page table. If the virtual page number is present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated, the same way as it would be from the page table itself.

The interesting case is what happens when the virtual page number is not in the TLB. The MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus if that page is used again soon, the second time it will result in a hit rather than a miss. When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there. When the TLB is loaded from the page table, all the fields are taken from memory.

Software TLB Management

Up until now, we have assumed that every machine with paged virtual memo-ry has page tables recognized by the hardware, plus a TLB. In this design, TLB management and handling TLB faults are done entirely by the MMU hardware.

Traps to the operating system occur only when a page is not in memory.

In the past, this assumption was true. However, some modern RISC ma-chines, including the MIPS, Alpha, and HP PA, do nearly all of this page manage-ment in software. On these machines, the TLB entries are explicitly loaded by the operating system. When a TLB miss occurs, instead of the MMU just going to the page tables to find and fetch the needed page reference, it just generates a TLB fault and tosses the problem into the lap of the operating system. The system must find the page, remove an entry from the TLB, enter the new one, and restart the instruction that faulted. And, of course, all of this must be done in a handful of instructions because TLB misses occur much more frequently than page faults.

Surprisingly enough, if the TLB is reasonably large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be quite efficient. The main gain here is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance.

Various strategies have been developed to improve performance on machines that do TLB management in software. One-approach attacks both reducing TLB misses and reducing the cost of a TLB miss when it does occur. To reduce TLB misses, sometimes the operating system can use its intuition to figure out which pages are likely to be used next and to preload entries for them in the TLB. For example, when a client process does an RPC to a server process on the same machine, it is very likely that the server will have to run soon. Knowing this, while processing the trap to do the RPC, the system can also check to see where the server's code, data, and stack pages are, and map them before they can cause TLB faults.

The normal way to process a TLB miss, whether in hardware or in software, is to go to the page table and perform the indexing operations to locate the page referenced. The problem with doing this search in software is that the pages hold-ing the page table may not be in the TLB, which will cause additional TLB faults during the processing. These faults can be reduced by maintaining a large (e.g., 4K) software cache of TLB entries in a fixed location whose page is always kept in the TLB. By first checking the software cache, the operating system can substantially reduce TLB misses.

4.4.4 Inverted Page Tables

Traditional page tables of the type described so far require one entry per vir-tual page, since virtual page number indexes them. If the address space consists of 232 bytes, with 4096 bytes per page, then over 1 million page table entries are needed. As a bare minimum, the page table will have to be at least 4 megabytes. On larger systems, this size is probably double.

However, as 64-bit computers become more common, the situation changes drastically. If the address space is now 254 bytes, with 4K pages, we need over 1015 bytes for the page table. Tying up 1 million gigabytes just for the page table is not double, not now and not for decades to come, if ever. Consequently, a different solution is needed for 64-bit paged virtual address spaces.

One such solution is the inverted page table. In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space. For example, with 64-bit virtual addresses, a 4K page, and 32 MB of RAM, an inverted page table only requires 8192 entries. The entry keeps track of which (process, virtual page) is located in the page frame.

Although inverted page tables save vast amounts of space, at least when the virtual address space is much larger than the physical memory, they have a serious downside: virtual-to-physical translation becomes much harder. When process n references virtual page p, the hardware can no longer find the physical page by using p as an index into the page table. Instead, it must search the entire inverted page table for an entry («, p). Furthermore, this search must be done on every memory reference, not just on page faults. Searching an 8K table on every memory reference is not the way to make your machine blindingly fast.

The way out of this dilemma is to use the TLB. If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page table>. On a TLB miss, however, the inverted page table has to be searched. Using a hash table as an index into the inverted page table, this search can be made reasonably fast, however." Inverted page tables are currently used on some IBM and Hewlett-Packard workstations and will become more common as 64-bit machines become widespread.

4.4 **Check Your Progress.**

Fill in the blanks.

1. The basic idea behind ___________ is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it.
2. Virtual memory can also work in a ___________ system.
3. Program-generated addresses are called ___________.

4.5 PAGE REPLACEMENT ALGORITHMS

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page that is to be removed has been modified in memory, it must be rewritten to: the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., a page contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

While it would be possible to pick a random page to replace at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms.

4.5.1 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced.

The optimal page algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible. Computers, like people, try to put off unpleasant events for as long as they can.

The only problem with this algorithm is that it is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next. (We saw a similar situation earlier with the shor-test job first scheduling algorithm-how can the system tell which job is shor-test?) Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the second run by using the page reference information collected during the first run.

In this way it is possible to compare the performance of realizable algorithms with the best possible one. If an operating system achieves a performance of, say, only 1 percent worse than the optimal algorithm, effort spent in looking for a better algorithm will yield at most a 1 percent improvement.

To avoid any possible confusion, it should be made clear that this log of page references refers only to the one program just measured. The page replacement algorithm derived from it is thus specific to that one program. Although this method is useful for evaluating page replacement algorithms, it is of no use in practical systems. Below we will study algorithms that are useful on real systems.

4.5.2 The Not Recently Used Page Replacement Algorithm

In order to allow the operating system to collect useful statistics about which pages are being used and which ones are not, most computers with virtual memory have two status bits associated with each page. R is set whenever the page is referenced (read or written). M is set when the page is written to (i.e., modified). The bits are contained in each page table entry. It is important to realize that these bits must be updated on every memory reference, so it is essential that they be set by the hardware. Once a bit has been set to 1, it stays 1 until the operating system resets it to 0 in software.

If the hardware does not have these bits, they can be simulated as follows. When a process is started up, all of its page table entries are marked as not in memory. As soon as any page is referenced, a page fault will occur. The operating system then sets the R bit (in its internal tables), changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction. If the page is subsequently written on, another page fault will occur, allowing the operating system to set the M bit and change the page's mode to READ/WRITE.

The R and M bits can be used to build a simple paging algorithm as follows. When a process is started up, both page bits for all its pages are set to 0 by the operating system. Periodically (e.g., on each clock interrupt), the R bit is cleared, to distinguish pages that have not been referenced recently from those that have been.

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their R and M bits. Class 0 : not referenced, not modified. Class 1 : not referenced, modified. Class 2 : referenced, not modified. Class 3: referenced, modified. Although class I pages seem, at first glance, impossible, they occur when a class 3 page has its R bit cleared by a clock interrupt. Clock interrupts do not clear the M bit because this information is needed to know whether the page has to be rewritten to disk or not.

The NRU (Not Recently Used) algorithm removes a page at random from the lowest numbered nonempty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick (typically 20 msec) than a clean page that is in heavy use. The main attraction of NRU is that it is easy to understand, efficient to implement, and gives a performance that, while certainly not optimal, is often adequate.

4.5.3 The First-In, First-Out (FIFO) Page Replacement Algorithm

Another low-overhead paging algorithm is the FIFO (First-In, First-Out) algorithm. To illustrate how this works, consider a supermarket that has enough shelves to display exactly k different products. One day, some company introduces a new convenience food-instant, freeze-dried, organic yogurt that can be reconstituted in a microwave oven. It is an immediate success, so our finite supermarket has to get rid of one old product in order to stock it.

One possibility is to find the product that the supermarket has been stocking the longest (i.e., something it began selling 120 years ago) and get rid of it on the grounds that no one is interested any more. In effect, the supermarket maintains a linked list of all the products it currently sells in the order they were introduced. The new one goes on the back of the list; the one at the front of the list is dropped.

AS a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list. When applied to stores, FIFO might remove mustache wax, but it might also remove flour, salt, or butter. When applied to computers the same problem arises. For this reason, FIFO in its pure form is rarely used.

4.5.4 The Second Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the R bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

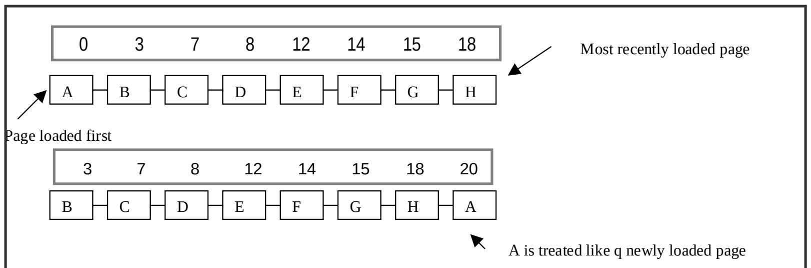
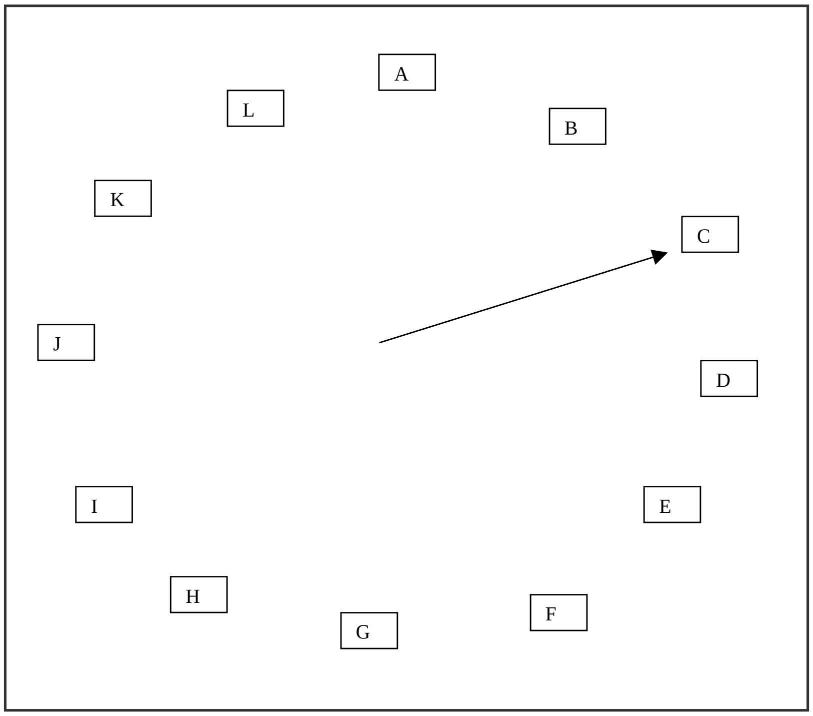


Figure 4-12.Operation of second chance, (a) Pages sorted in FIFO order, (b) Page list if a page fault occurs at lime 20 and A has its A bit set

4.5.5 The Clock Page Replacement Algorithm

Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is constantly moving pages around on its list. A better approach is to keep all the pages on a circular list in the form of a clock, as shown in Figure. 4.13. A hand points to the oldest page.



R = 0: Evict the page

R = 1: Clear R and advance hand

Figure 4.13- The clock page replacement algorithm.

When a page fault occurs, the page being pointed to by the hand is inspected. If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If R is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R - 0. Not surprisingly, this algorithm is called clock. It differs from second chance only in the implementation.

4.5.6 The Least Recently Used (LRU) Page Replacement Algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for

When a page fault occurs, the page tje hand is pointing to is inspected. The action taken depends on the R bit.

Ages will probably, remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called LRU (Least Recently Used) paging.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Find-ing a page in the list, deleting it, and then moving it to the front is a very time con-suming operation, even in hardware (assuming that such hardware could be built).

However, there are other ways to implement LRU with special hardware. Let us consider the simplest way first. This method requires equipping the hardware with a 64-bit counter, C, that is automatically incremented after each instruction. Furthermore, each page table entry must also have a field large enough to contain the counter. After each memory reference, the current value of C is stored in the page table entry for the page just referenced. When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.

Now let us look at a second hardware LRU algorithm. For a machine with n page frames, the LRU hardware can maintain a matrix of nxn bits, initially all zero. Whenever page frame k is referenced, the hardware first sets all the bits of row k to 1, and then sets all the bits of column k to 0. At any instant, the row whose binary value is lowest is the least recently used; the row whose value is next lowest is next least recently used, and so forth. The workings of this algorithm are given in Figure. 4.14 for four page frames and page references in the order

0123210323

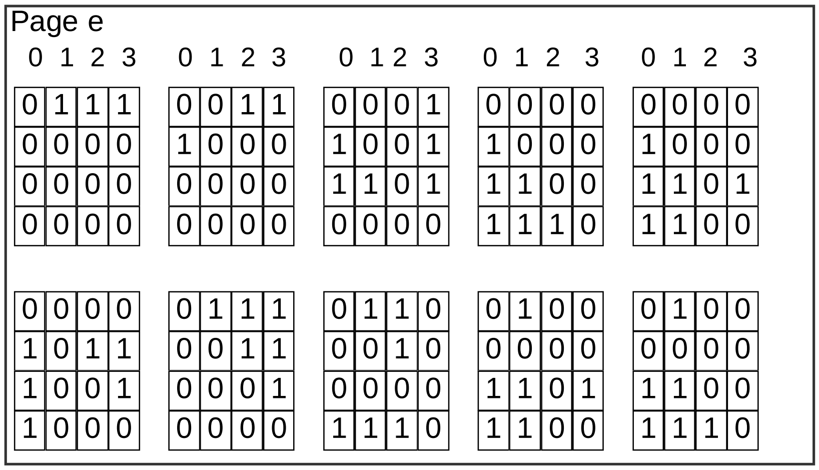


Figure 4.14 Least recently used Algorithm.

4.5 **Check Your Progress.**

Fill in the blanks.

1. The best possible ______________ algorithm is easy to describe but impossible to implement
2. The ______________ algorithm removes a page at random from the lowest numbered nonempty class.
3. Another ______________ paging algorithm is the FIFO (First-In, First-Out) algorithm.

4.6 DESIGN ISSUES FOR PAGING SYSTEMS

In the previous sections we have explained how paging works and have given a few of the basic page replacement algorithms. But knowing the bare mechanics is not enough. To design a system, you have to know a lot more to make it work well. It is like the difference between knowing how to move the rook, knight, bishop, and other pieces in chess, and being a good player. In the following sec-tions, we will look at other issues that operating system designers must consider carefully in order to get good performance from a paging system.

4.6.1 The Working Set Model

In the purest form of paging, processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruc-tion. Other page faults for global variables and the stack usually follow quickly. After a while, the process has most of the pages it needs and settles down to run with relatively few page faults. This strategy is called demand paging because pages are loaded only on demand, not in advance.

Of course, it is easy enough to write a test program that systematically reads all the pages in a large address space, causing so many page faults that there is not enough memory to hold them all. Fortunately, most processes do not work this way. They exhibit a locality of reference, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multi-pass compiler, for example, references only a fraction of all the pages, and a different fraction at that.

The set of pages that a process is currently using is called its working set If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase (e.g., the next pass of the compiler). If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly since executing an instruction often takes a few nanoseconds and reading in a page from the disk typically takes tens of milliseconds. At a rate of one or two instructions per 20 milliseconds, it will take ages to finish. A program caus-ing page faults every few instructions is said to be thrashing.

In a timesharing system, processes are frequently moved to disk (i.e., all their pages are removed from memory) to let other processes have a turn at the CPU.

The question arises of what to do when a process is brought back in again. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having 2)0, 50, or even 100 page faults every time a process is loaded is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault.

Therefore, many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the working set model It is designed to greatly reduce the page fault rate. Loading the pages before, letting processes run is also called pre-paging.

To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. One way to monitor this in-formation is to use the aging algorithm discussed above. Any page containing a 1 bit among the high order n bits of the counter is considered to be a member of the working set. If a page has not been referenced in n consecutive clock ticks, it is dropped from the working set. The parameter n has to be determined experimen-tally for each system, but the system performance is usually not especially sensi-tive to the exact value.

Information about the working set can be used to improve the performance of the clock algorithm. Normally, when the hand points to a page whose R bit is 0, the page is evicted. The improvement is to check to see if that page is part of the working set of the current process. If it is, the page is spared. This algorithm is called wsclock.

4.6.2 Local versus Global Allocation Policies

In the preceding sections we have discussed several algorithms for choosing a page to replace when a fault occurs. A major issue associated with this choice (which we have carefully swept under the rug until now) is how memory should be allocated among the competing runnable processes.

Take a look at Figure. 4.15(a). In this figure, three processes. A, B, and C, make up the set of runnable processes. Suppose A gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to A, or should it consider all the pages in memory? If it looks only at A's pages, the page with the lowest age value is A5, so we get the situation of Figure. 4.15(b).

On the other hand, if the page with the lowest age value is removed without regard to whose page it is, page will be chosen and we will get the situation of Figure 4.15(c). The algorithm of Figure 4.15(b) is said to be a local page replacement algorithm, whereas Figure. 4-15(c) is said to be a global algorithm. Local algorithms effectively correspond to allocating every process a fixed fraction of the memory.

0	10	A0	A0
A1	7	A1	A1
A2	5	A2	A2
A3	4	A3	A3
A4	6	A4	A4
A5	3	A6	A5
B0	9	B0	B0
B1	4	B1	B1
B2	6	B2	B2
B3	2	B3	A6
B4	5	B4	B4
B5	6	B5	B5
B6	12	B6	B6
C1	3	C1	C1
C2	5	C2	C2
C3	6	C3	C3
(a)		**(b)**	**(c)**

Figure 4.15 Local versus global page replacement, (a) Original configuration, (b) Local page replacement, (c) Global page replacement.

Global algorithms dynamically allocate page frames among the manageable proc-esses. Thus the number of page frames assigned to each process varies in time. In general, global algorithms work better, especially when the working set size can vary over the lifetime of a process. If a local algorithm is used and the working set grows, thrashing will result, even if there are plenty of free page frames. If the working set shrinks, local algorithms waste memory. If a global al-gorithm is used, the system must continually decide how many page frames to assign to each process. One way is to monitor the working set size as indicated by the aging bits, but this approach does not necessarily prevent thrashing. The working set may change size in microseconds, whereas the aging bits are a crude measure spread over a number of clock ticks.

Another approach is to have an algorithm for allocating page frames to processes. One way is to periodically determine the number of running processes and allocate each process an equal share. Thus with 475 available (i.e., non-operating system) page frames and 10 processes, each process gets 47 frames. The remaining 5 go into a pool to be used when page faults occur.

Although this method seems fair, it makes little sense to give equal shares of the memory to a 10K process and a 300K process. Instead, pages can be allocated in proportion to each process' total size, with a 300K process getting 30 times the allotment of a 10K process. It is probably wise to give each process some minimum number, so it can run, no matter how small it is. On some machines, for example, a single instruction may need as many as six pages because the instruction itself, the source operand, and the destination operand may all straddle page boundaries. With an allocation of only five pages, programs containing such instructions cannot execute at all.

Neither the equal allocation nor the proportional allocation method directly deals with the thrashing problem. A more direct way to control it is to use the Page Fault Frequency or PFF allocation algorithm. For a large class of page replacement algorithms, including LRU, it is known that the fault rate decreases as more pages are assigned, as we discussed above. This property is illustrated in Figure. 4.15

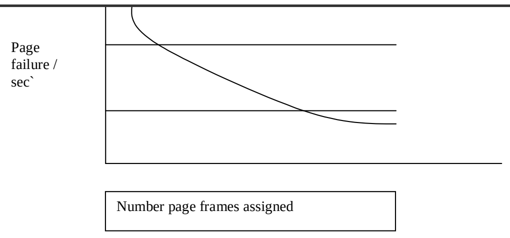


Figure 4.15 - Page fault of the number of page frame

The dashed line marked A corresponds to a page fault rate that is unacceptably high, so the faulting process is given more page frames to reduce the fault rate. The dashed line marked B corresponds to a page fault rate so low that it can be concluded that the process has too much memory. In this case page frames may be taken away from it. Thus, PFF tries to keep the paging rate within acceptable bounds.

If it discovers that there are so many processes in memory that it is not possible to keep all of them below A, then some process is removed from memory, and its page frames are divided up among the remaining processes or put into a pool of available pages that can be used on subsequent page faults. The decision to remove a process from memory is a form of load control. It shows that even with paging, swapping is still needed, only now swapping is used to reduce potential demand for memory, rather than to reclaim blocks of it for immediate use.

4.6.3 Page Size

The page size is often a parameter that can be chosen by the operating system. Even if the hardware has been designed with, for example, 512-byte pages, the operating system can easily regard pages 0 and 1, 2 and 3, 4 and 5, and so on, as 1K pages by always allocating two consecutive 512-byte page frames for them. Determining the optimum page size requires balancing several competing fac-tors. To start with, a randomly chosen text, data, or stack segment will not fill an integral number of pages. On the average, half of the final page will be empty. The extra space in that page is wasted this wastage is called internal fragmenta-tion. With n segments in memory and a page size of *p* bytes, *np/2* bytes will be wasted on internal fragmentation. This reasoning argues for a small page size.

Another argument for a small page size becomes apparent if we think about a program consisting of eight sequential phases of 4K each. With a 32K page size, the program must be allocated 32K all the time. With a 16K page size, it needs only 16K. With a page size of 4K or smaller, it requires only 4K at any instant. In general, a large page size will cause more unused program to be in memory than a small page size.

On the other hand, small pages mean that programs will need many pages, hence a large page table. A 32K program needs only four 8K pages, but 64 512-byte pages. Transfers to and from the disk are generally a page at a time, with most of the time being for the seek and rotational delay, so that transferring a small page takes almost as much time as transferring a large page. It might take 64x 15 msec to load 64 512-byte pages, but only 4x25 msec to load four 8K pages.

On some machines, the page table must be loaded into hardware registers every time the CPU switches from one process to another. On these machines having a small page size means that the time required to load the page registers gets longer as the page size gets smaller. Furthermore, the space occupied by the page, table increases as the page size decreases.

This last point can be analyzed mathematically. Let the average process size be s bytes and the page size be *p* bytes. Furthermore, assume that each page entry requires e bytes. The approximate number of pages needed per process is then s/p, occupying *se/p* bytes of page table space. The wasted memory in the last page of the process due to internal fragmentation is p/2. Thus, the total overhead due to the page table and the internal fragmentation loss is given by

overhead = *se/p* + *p/2*

The first term (page table size) is large when the page size is small. The second term (internal fragmentation) is large when the page size is large. The optimum must lie somewhere in between. By taking the first derivative with respect to p and equating it to zero, we get the equation

$$-se/p2+ 1/2 = 0$$

From this equation we can derive a formula that gives the optimum page size (considering only memory wasted in fragmentation and page table size).

For s = 128K and e = 8 bytes per page table entry, the optimum page size is 1448 bytes. In practice 1K or 2K would be used, depending on the other factors (e.g., disk speed). Most commercially available computers use page sizes ranging from 512 bytes to 64K.

Up until now, our whole discussion has assumed that virtual memory is trans-parent to processes and programmers, that is, all they see is a large virtual address space on a computer with a smaller physical memory. With many systems, that is true, but in some advanced systems, programmers have some control over the memory' map and can use it in nontraditional ways. In this section, we will briefly look at a few of these.

One reason for giving programmers control over their memory map is to allow two or more processes to share the same memory. If programmers can name regions of their memory, it may be possible for one process to give another process the name of a memory region so that process can also map it in. With two (or more) processes sharing the same pages, high bandwidth sharing becomes possible-one process writes into the shared memory and another one reads from it.

Sharing of pages can also be used implement a high-performance mes-sage-passing system. Normally, when messages-are passed, the data are copied from one address space to another, at considerable cost. If processes can control their page map, having the sending process unmapped the page(s) containing the message, and the receiving process mapping them in can pass a message. Here only the page names have to be copied, instead of all the data.

Yet another advanced memory management technique is distributed shared memory. The idea here is to allow multiple processes over a network to share a set of pages, possibly, but not necessarily, as a single shared linear address space. When a process references a page that is not currently mapped in, it gets a page fault. The page fault handlers, which may be in the kernel or in user space, then locates the machine holding the page and sends it a message asking it to unmapped the page and send it over the network. When the page arrives, it is mapped in and the faulting instruction is restarted.

4.6 Check Your Progress.

Fill in the Blanks.

1. The set of pages that a process is currently using is called its ___________.
2. Many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the __________.
3. A ___________ has other advantages besides simplifying the handling of data structures that are growing or shrinking
4. ___________ also facilitates sharing procedures or data between several processes..

4.7 SEGMENTATION

The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another. For many problems, having two or more separate virtual address spaces may be much better than having only one. For example, a compiler has many tables that are built up as compilation proceeds, possibly including

1. The source text being saved for the printed listing (on batch systems).
2. The symbol table, containing the names and attributes of variables.
3. The table containing all the integer and floating-point constants used.
4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space, as in Figure - 4.16.

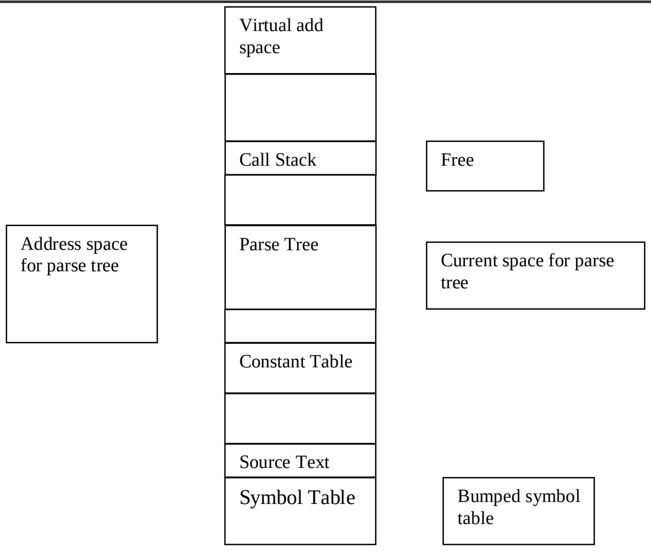


Figure 4.16 Segmentation

Virtual address space Address space allocated to the parse tree.

Consider what happens if a program has an exceptionally large number of variables. The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables. The compiler could, of course, simply issue a message saying that the compilation cannot continue due to too many variables, but doing so does not seem very sporting when unused space is left in the other tables.

Another possibility is to play Robin Hood, taking space from the tables with an excess of room and giving it to the tables with little room. This shuffling can be done, but it is analogous to managing one's own overlays-a nuisance at best and a great deal of tedious, unrewarding work at worst. What is really needed is a way of freeing the programmer from having to manage the expanding and contracting tables, in the same way that virtual memory eliminates the worry of organizing the program into overlays.

A straightforward and extremely general solution is to provide the machine with many completely independent address spaces, called segments. Each segment consists of a linear sequence of addresses, from 0 to some maximum. The length of each segment may be anything from 0 to the maximum allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.

Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up but segments are usually very large, so this occurrence is rare. To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment. A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking. If each procedure occupies a sep-arate segment, with address 0 as its starting address, the linking up of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment n will use the two-part address (n, 0) to address word 0 (the entry point).

If the procedure in segment n is subsequently modified and recompiled, no other procedures need be changed (because no starling addresses have been modified), even if the new version is larger than the old one. With a one-dimensional memory, the procedures are packed tightly next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of other, unrelated procedures. This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses. If a program contains hundreds of procedures, this process can be costly.

Segmentation also facilitates sharing procedures or data between several processes. A common example is the shared library. Modern workstations that run advanced window systems often have extremely large graphical libraries compiled into nearly every program. In a segmented system, the graphical library can be put in a segment and shared by multiple processes, eliminating the need for having it in every process' address space. While it is also possible to have shared libraries in pure paging systems, it is much more complicated. In effect, these systems do it by simulating segmentation.

Because each segment forms a logical entity of which the programmer is aware, such as a procedure, or an array, or a stack, different segments can have different kinds of protection. A procedure segment can be specified as execute only, prohibiting attempts to read from it or store into it. A floating-point array can be specified as read/write but not execute, and attempts to jump to it will be caught. Such protection is helpful in catching programming errors.

You should try to understand why protection makes sense in a segmented memory but not in a one-dimensional paged memory. In a segmented memory the user is aware of what :s in each segment. Normally, a segment would not contain a procedure and a stack, for example, but one or the other. Since each segment contains only one type of object, the segment can have the protection appropriate for that particular type. Paging and segmentation are compared in Figure. 4.17

Consideration	Paging	Segmentation
Need the programmer be aware ? that this technique is being used	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address without having to buy memory and Protectio	To allow programs and data to be broken up into logically independent address spaces and to aid sharing

Figure 4.17 - Comparisonof paging and segmentation

That paging was invented to eliminate. Because the user of a segmented memory has the illusion that all segments are in main memory all the time-that is, he can address them as though they were-he can protect each segment separately, with-out having to be concerned with the administration of overlaying them.

4.7.1 Implementation of Pure Segmentation

The implementation of segmentation differs from paging in an essential way: pages are fixed size and segments are not.

4.7.2 Segmentation With Paging : MULTICS

If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory in their entirety. This leads to the idea of paging them, so that only those pages that are actually needed have to be around. Several signifi-cant systems have supported paged segments. In this section we will describe the first one: MULTICS. In the next one we will discuss a more recent one: the Intel Pentium.

MULTICS ran on the Honeywell 6000 machines and their descendants and pro-vided each program with a virtual memory of up to 2k segments (more than

250,000), each of which could be up to 65,536 (36-bit) words long. To implement this, the MULTICS designers chose to treat each segment as a virtual memory and to page it, combining the advantages of paging (uniform page size and not having to keep the whole segment in memory if only part of it is being used) with the advantages of segmentation (ease of programming, modularity, protection, and sharing).

Each MULTICS program has a segment table, with one descriptor per segment-Since there are potentially more than a quarter of a million entries in the table, the segment table is itself a segment and is paged. A segment descriptor contains an indication of whether the segment is in main memory or not. If any part of the segment is in memory, the segment is considered to be in memory, and its page table will be in memory. If the segment is in memory, its descriptor contains an 18-bit pointer to its page table see Figure. 4.18. Because physical addresses are 24 bits and pages are aligned on 64-byte boundaries (implying that the low-order 6 bits of page addresses are 000000), only 18 bits are needed in the descriptor to store a page table address. The descriptor also contains the segment size, the protection bits, and a few other items. Figure 4.18 illustrates a MULTICS segment descriptor. The address of the segment in secondary memory is not in the segment descriptor but in another table used by the segment fault handler.

1 .		
Segment		6 descriptor
Segment		5 descriptor
Segment		4 descriptor
Segment		3 descriptor
Segment		2 descriptor
Segment		1 descriptor
Segment		0 descriptor
1		1
	Page 2 entry	
	Page e1	Ntry
	PageO	Ntry
Pac	Table	for segment 3
I		
	Page 2	Entry
	Pa	Entry
	Pa	Entry

Page table for segment 1

Page size:

0 = 1024 words -

1 = 64 words

Protection bits

Figure 4.18 The MULTICS virtual memory (a) The descriptor segment points to the page tables, (b) A segment descriptor. The numbers are the field lengths.

Each segment is. an ordinary virtual address space and is paged in the same way as the nonsegmented paged memory described earlier in this chapter. The normal page size is 1024 words (although a few small segments used by MULTICS itself are not paged or are paged in units of 64 words to save physical memory).

An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page, as shown in Figure. 4.19. When a memory reference occurs, the following algorithm is carried out.

1. The segment number is used to find the segment descriptor.

2. A check is made to see if the segment's page table is in memory. If the page table is in memory, it is located. If it is not, a segment fault occurs.

3. The page table entry for the requested virtual page is examined. If the page is not in memory, a page fault occurs. If it is in memory, the main memory address of the start of the page is extracted from the page table entry.

4. The offset is added to the page origin to give the main memory address where the word is located.

5. The read or store finally takes place. Address within the segment

Segment number

Page number	Offset within the page
18	6

Figure 4.19 - A 34-bit multicast virtual address.

For simplicity, the fact that the de-scriptor segment is itself paged has been omitted. What really happens is that a register (the descriptor base register), is used to locate the descriptor segment's page table, which, in turn, points to the pages of the descriptor segment. Once the descriptor for the needed segment has been found, the addressing procedures are as shown in Figure. 4.20.

As you have no doubt guessed by now, if the preceding algorithm were actu-ally carried out by the operating system on every instruction, programs would not run very fast. In reality, the MULTICS hardware contains a 16-word high-speed TLB that can search all its entries in parallel for a given key. It is illustrated in Figure. 4.20. When an address is presented to the computer, the addressing hardware first checks to see if the virtual address is in the TLB. If so, it gets the page frame number directly from the TLB and forms the actual address of the referenced word without having to look in the descriptor segment or page table.

The addresses of the 16 most recently referenced pages are kept in the TLB. Programs whose working set is smaller than the TLB size will come to equilibrium with the addresses of the entire working set in the TLB and therefore will

SEGMENTATION

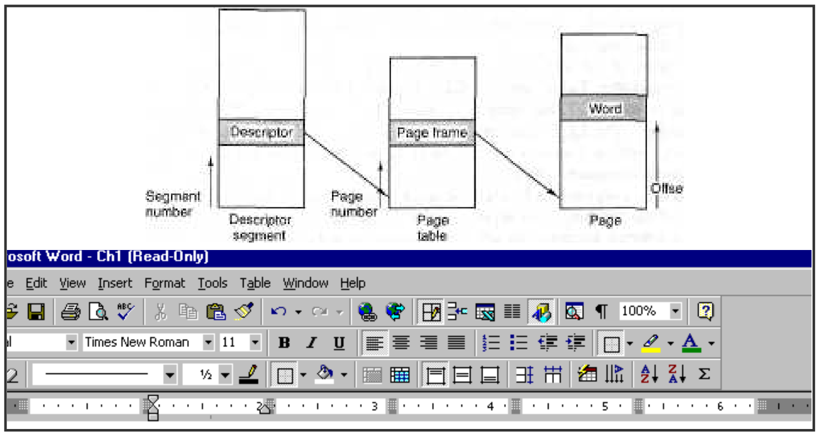


Figure 4.20 - A simplified version of the MULTICS TLB

run efficiently. If the page is not in the TLB, the descriptor and page tables are actually referenced to find the page frame address, and the TLB is updated to in-clude this page, the least recently used page being thrown out. The age field keeps track of which entry is the least recently used. The reason that a TLB is used is for comparing the segment and page number of. all the entries in parallel.

4.7.3 Segmentation with Paging: The Intel Pentium

In many ways, the virtual memory on the Pentium (and Pentium Pro) resem-bles MULTICS, including the presence of both segmentation and paging. Whereas MULTICS has 256K independent segments, each up to 64K 36-bit words, the Pen-tium has 16K independent segments, each holding up to 1 billion 32-bit words. Although there are fewer segments, the larger segment size is far more important, as few programs need more than 1000 segments, but many programs need seg-ments holding megabytes.

The heart of the Pentium virtual memory consists of two tables, the LDT (Local Descriptor Table) and the GDT (Global Descriptor Table). Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

To access a segment, a Pentium program first loads a selector for that segment into one of the machine's six segment registers. During execution, the CS register holds the selector for the code segment and the DS register holds the selector for the data segment. The other segment registers are less important. Each selector is a 16-bit number.

One of the selector bits tells whether the segment is local or global (i.e., whether it is in the LDT or GDT). Thirteen other bits specify the LDT or GDT entry number, so these tables are each restricted to holding 8K segment descriptors. The other 2 bits relate to protection, and will be described later. Descriptor 0 is forbidden. It may be safely loaded into a segment register to indicate that the segment register is not currently available. It causes a trap if used.

At the time a selector is loaded into a segment register, (he corresponding de-scriptor is fetched from the LDT or GDT and stored in micro-program registers, so it can be accessed quickly. A descriptor consists of 8 bytes, including the segment's base address, size, and other information.

The format of the selector has been cleverly chosen to make locating the descriptor easy. First either the LDT or GDT is selected, based on selector bit 2. Then the selector is copied to an internal scratch register, and the 3 low-order bits set to 0. Finally, the address of either the LDT or GDT table is added to it, to give a direct pointer to the descriptor. For example, selector 72 refers to entry 9 in the GDT, which is located at address GDT + 72.

Let us trace the steps by which a (selector, offset) pair is converted to a physi-cal address. As soon as the micro-program knows which segment register is being used, it can find the complete descriptor corresponding to that selector in its internal registers. If the segment does not exist (selector 0), or is currently paged out, a trap occurs.

It then checks to see if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should simply be a 32-bit field in the descriptor giving the size of the segment, but there are only 20 bits available, so a different scheme is used. If the Gblt (Granularity) field is 0, the Limit field is the exact segment size, up to 1 MB. If it is 1, the Limit field gives the segment size in pages instead of bytes. The Pentium page size is fixed at 4K bytes, so 20 bits are enough for segments up to 232 bytes.

Assuming that the segment is in memory and the offset is in range, the Pentium then adds the 32-bit Base field in the descriptor to the offset to form what is called a linear address. The Base field is broken up into three pieces and spread all over the descriptor for compatibility with the 286, in which the Base is only 24 bits. In effect, the Base field allows each segment to start at an arbitrary place within the 32-bit linear address space

If paging is disabled (by a bit in a global control register), the linear address is interpreted as the physical address and sent to the memory for the read or write. Thus with paging disabled, we have a pure segmentation scheme, with each segment's base address given in its descriptor. Segments are permitted to overlap, incidentally, probably because it would be too much trouble and take too much time to verify that they were all disjoint.

On the other hand, if paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables, pretty much as in our earlier examples. The only real complication is that with a 32-bit virtual address and a 4K page, a segment might contain 1 million pages, so a two-level mapping is used to reduce the page table size for small segments.

Each running program has a page directory consisting of 1024 32-bit curies. It is located at an address pointed to by a global register. Each entry in this directory points to a page table also containing [024 32-bit entries. The page table entries point to page frames. Then the Page field is used as an index into the page table to find the physical address of the page frame. Finally, Off is added to the address of the page frame to get the physical address of the byte or word needed.

The page table entries are 32 bits each, 20 of which contain a page frame number. The remaining bits contain access and dirty bits, set by the hardware for the benefit of the operating system, protection bits, and other utility bits.

Each page table has entries for 1024 4K-page frames, so a single page table handles 4 megabytes of memory. A segment shorter than 4M will have a page di-rectory with a single entry, a pointer to its one and only page table. In this way, the overhead for short segments is only two pages, instead of the million pages that would be needed in a one-level page table.

To avoid making repeated references to memory, the Pentium, like MULTICS, has a small TLB that directly maps the most recently used Dir-Page combina-tions onto the physical address of the page frame. Only when the current combination is not present in the TLB is the mechanism actually carried out and the TLB updated.

A little thought will reveal the fact that when paging is used, there is really no point in having the Base field in the descriptor be nonzero. That entire Base does is cause a small offset to use an entry in the middle of the page directory, instead of at the beginning. The real reason for including Base at all is to allow pure (non-paged) segmentation, and for compatibility with the 286, which always has paging disabled (i.e., the 286 has only pure segmentation, but not paging).

It is also worth noting that if some application does not need segmentation but is content with a single, paged, 32-bit address space, that model is possible. All the segment registers can be set up with the same selector, whose descriptor has Base - 0 and Limit set to the maximum. The instruction offset will then be the' linear address, with only a single address space used-in effect, normal paging.

All in all, one has to give credit to the Pentium designers. Given the conflict-ing goals of implementing pure paging, pure segmentation, and paged segments, while at the same time being compatible with the 286, and doing all of this efficiently, the resulting design is surprisingly simple and clean.

4.8 SUMMARY

In this chapter we will investigate a number of different memory management schemes, ranging from very simple to highly sophisticated. We will start at the beginning and look first at the simplest possible memory management system and then gradually progress to more and more elaborate ones.

Sources : *http://ecee.colorado.edu, http://www.cs.vu.nl (E-book)*

4.9 CHECK YOUR PROGRESS - ANSWERS

4.2

1. RAM	2. Partitions	3. Small jobs

4.3

1. Swapping,	2. Holes	3. Swap area

4.4

1. Virtual memory
2. Multiprogramming
3. Virtual addresses

4.5

1. Page replacement
2. NRU (Not Recently Used)
3. Low-overhead

4.6

1.	Working set	2.	Working set model
3.	Segmented memory	4.	Segmentation

4.10 QUESTIONS FOR SELF - STUDY

1. Describe swapping and virtual memory.
2. Explain various algorithms used for page replacement.
3. Describe paging in virtual memory.
4. Explain segmentation with paging.

4.11 SUGGESTED READINGS

1. **Operating System Concepts** By Abraham Silberschatz, Peter B. Galvin & Greg Gagne.

2. **Operating systems** By Stuart E. Madnick, John J. Donovan

Notes

Notes

Chapter : 5

Information Management

5.0 OBJECTIVES

After studying this chapter you will be able to -

- Analyze simple file system
- Discuss General Model of a file system
- Explain symbolic file system and Basic file system
- Describe various access control verification processes.

5.1 INTRODUCTION

The modules of information management are sometimes collectively referred to as the file system. Information management is conceptually simple. Yet informa-tion is one of the most important resources of an operating system and perhaps the one most poorly managed in contemporary systems.

In Figure 5-1 we placed the modules of information management in a single outer ring of the kernel of the operating system. However, a full implementation of this manager can be quite complicated. Thus our approach is to further delineate the modules of the information manager into levels, as shown in Figure 5.1. Such delineation was first presented by and is a framework for imple-menting a file system.

Each level of the file system follows structured programming techniques. Each level depends only on levels below it and only calls downward. This hierarchical approach for the design of the file system does not require that the rest of the operating system be implemented in the hierarchical manner implied in Fig 5.1.

The file system is intended to provide convenient management of both permanent (long-term) and temporary (short-term) information. The programmer is freed from problems such as the allocation of space for his information, physical storage for-mats, and I/0 accessing. One goal of a file system is to allow the programmer to concern himself only with the logical structure and operations performed in process-ing his information. File systems may also make it possible to share information among users and protect information from unauthorized access.

We will present only basic file system concepts in this chapter. It is useful to distinguish among file systems (i.e.. basic information management mechanisms), data management systems, and database systems.

A file system is concerned only with the simple logical organization of information. It deals with collections of unstructured and uninterpreted information at the operating system level. Each separately identified collection of information is called a file or data set.

A data management system performs some structuring but no interpretation of the information, e.g., the system may know that each me is divided into entries of a certain size and although it allows the programmer to manipulate these entries it doesn't know what they mean (IBM's IMS-II is a data management system).

A database system is concerned with both the structure and interpretations of data items. An example is an airline reservations system, which allows a user to ask, "How many people are there on flight 904?"

This chapter focuses mainly on the systems. A file is a collection of related infor-mation units (records). For example, in an inventory control application, one line of an invoice is a data item, a complete invoice is a record, and a complete set of such records is a file.

File systems have evolved from simple mechanisms for keeping track of system programs to more complex ones that provide users and the operating system with the means for storing, retrieving, and sharing information. Thus the file system is needed by other components of the operating system as well as by user jobs.

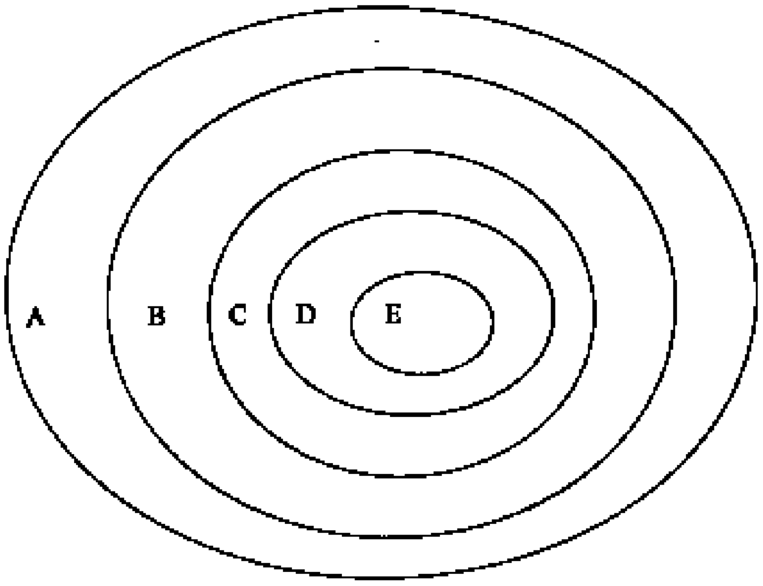


Figure 5.1 Kernel of Operating System

Let us first take a simple user request for information and analyze the steps necessary to fulfill that request. From this simple example we will construct a general model of a file system. We then apply that model to design more flexible file systems.

5.2 SIMPLE FILE SYSTEM

Let us consider the steps needed to process the following PL/I-like statement.

READ FILE (ETHEL) RECORD (4) INTO LOCATION (12000)

This is a request to read the fourth record of the file named ETHEL into a mem-ory location 12000. Such a request invokes elements of the file system.

For the purpose of this example, let us assume that the file ETHEL is stored on a disk, as shown in Figure 5.2The file ETHEL is the shaded portion and consists of seven logical records. Each logical record is 500 bytes long. The disk consists of 16 physical blocks of 1000 bytes each. (We call the physical units of the storage device blocks to differentiate from the logical units of interest to the programmer which are called records-these are analogous to PL/I data structures.) We can store two logical records of file ETHEL. into each physical block. Instead of using the physical two-component (track number, record number) address, we will assign each physical block a unique number, as indicated in Figure 5.2.

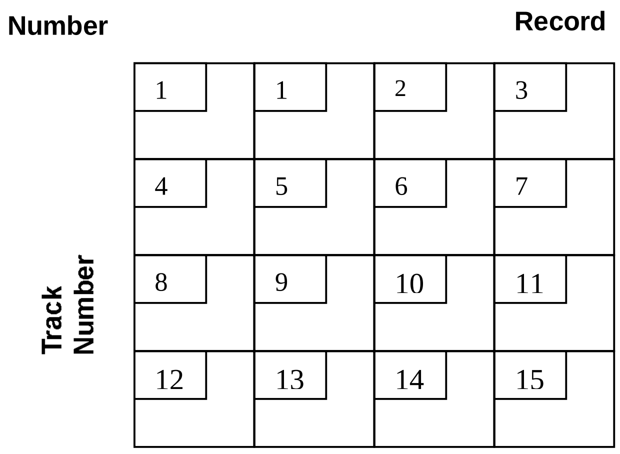


Figure 5.2 Physical File Storage

5.2.1 File Directory Database

Information on each existing file's status, for example, name and location, must be maintained in a table. As noted above, this table is often called a file directory or a Volume Table of Contents in many systems, most notably IBM's OS/360.

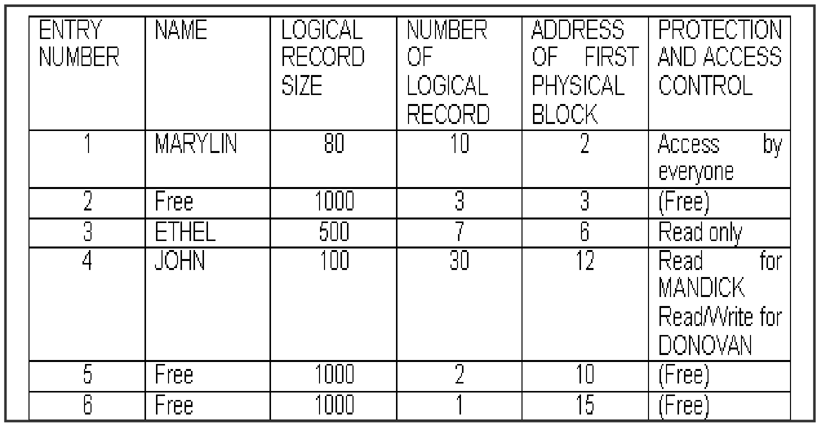
The name Volume Table of Contents comes from viewing the storage device medium as a "container of information" and thus a volume. The distinction between a storage device and a volume becomes important when removable volumes (e.g. disk pack, tape) are used. These volumes can be physically removed from their original storage device and placed on another storage device of the same type, sometimes even on a different computer system.

Figure 5-3 depicts a sample file directory. Each entry contains information nec-essary to find and access a file. The logical organization, i.e., programmer's view, is described in terms of the logical record length and the number of records.

The physical location of the file on the storage device can be determined by means of the starting block number information in conjunction with the logical record length and number of records. File ETHEL, for example, consists of seven SOD-byte records that require 3500 bytes of storage. Since the physical blocks are 1000 bytes long, four blocks are needed to hold files ETHEL. The first of these blocks, noted in the file directory, is 6. Thus, ETHEL occupies physical storage blocks 6, 7, 8, and 9, as previously depicted in Figure 5.2.

An important point here is that we distinguish between logical structuring and physical structuring. ETHEL is logically structured from the user's point of view as a sequential file, i.e., it consists of seven records numbered 1 through 7. It is also physically stored as a sequential file, i.e.. its records are stored adjacent to one another in consecutive physical blocks. However, the physical structure of ETHEL need not be sequential. The blocks of ETHEL could be scattered allover secondary storage, and a mapping between the logical records and the physical blocks might then become quite complicated. Examples of other such structuring are presented later in Sections.

There are six entries listed in the file directory of Figure 5.3. Three of these (MARILYN, ETHEL and JOHN)



ENTRY NUMBER	NAME	LOGICAL RECORD SIZE	NUMBER OF LOGICAL RECORD	ADDRESS OF FIRST PHYSICAL BLOCK	PROTECTION AND ACCESS CONTROL
1	MARYLIN	80	10	2	Access by everyone
2	Free	1000	3	3	(Free)
3	ETHEL	500	7	6	Read only
4	JOHN	100	30	12	Read for MANDICK Read/Write for DONOVAN
5	Free	1000	2	10	(Free)
6	Free	1000	1	15	(Free)

Figure 5.3 Sample File Directories

corresponds to real user files. The other three entries are needed merely to account for presently unused storage space. Thus the file directory indicates the usage of all the physical storage blocks. (Note: blocks 0 and 1 are reserved for special purposes to be explained later.)

5.2.2 Steps to Be Performed

Figure 5.4 presents the steps that must be performed to satisfy a request, such as:

READ FILE (ETHEL) RECORD (4) INTO LOCATION (I 2000)Q1

5.2 **Check Your Progress.**

Fill in the Blanks.

1. The name Volume Table of Contents comes from viewing the storage device medium as a ___________.
2. The physical location of the file on the storage device can be determined by means of the ____________.
3. ____________ Indicates the usage of all the physical storage blocks.

5.3 GENERAL MODEL OF A FILE SYSTEM

In this section a general model of a file system is introduced. Most of key components of a file system correspond to generalizations of specific steps of Figure 5.4.

The components in this file system are organized as a structured hierarchy. Each level in the hierarchy depends only on levels below it. This concept improves the possibility of systematic debugging procedures. For example, once the lowest level is debugged, changes at higher levels have no effect. Debugging can proceed from lowest to highest levels.

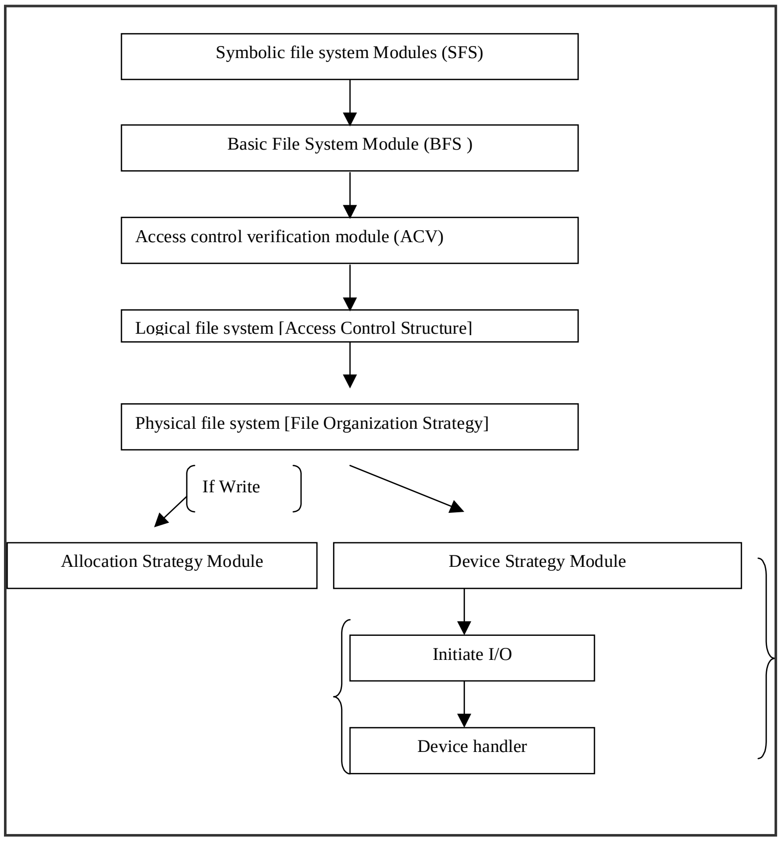

Symbolic file system Modules (SFS)
Basic File System Module (BFS)
Access control verification module (ACV)
Logical file system [Access Control Structure]
Physical file system [File Organization Strategy]
If Write
Allocation Strategy Module
Device Strategy Module
Initiate I/O
Device handler

FIGURE 5.4 GENERAL MODEL OF A FILE SYSTEM

Although the particular details presented in this chapter (e.g., format of file directory, file maps) may vary significantly from system to system, the basic structure is common to most contemporary me systems. For example, depending on the structure of a specific file system, certain of the modules in Figure 5.5 may be merged together, further subdivided, or even eliminated-but the underlying structure should still be the same.

5.3.1 File Directory Maintenance

Before discussing the functions of the file system components, it is important to answer certain questions about the file directory, such as: How are the file directory entries created and filled in? Where is the file directory stored? Is it necessary to search the entire directory for every request?

In a basic system such as IBM's Basic Operating System/360 (OS/360), the programmer must keep track of available storage space and maintain the me directory by control cards similar to:

CREATE ETHEL, RECSIZE=500, NUMRECS=7, LOCATION = 6 DELETE JOHN

The CREATE command adds a new entry to the file directory and the DELETE command removes an old one.

If the entire file directory is kept in main memory all the time, a significant amount of memory may be needed to store it. A more general approach is to treat the file directory as a file and place it on the storage volume. Furthermore, if the me directory is stored on the volume, the files may be easily transferred to another system by physically transferring the volume (tape reel, disk pack, etc.) containing these files as well as the corresponding me directory.

If the file directory is stored on the volume, then it may take a substantial amount of time to search it for each access to files. Although the directory may be quite large, possibly containing thousands of entries, usually only a few files are ever used at one time. Thus if we copy the entries for files that are currently in use into main memory, the subsequent search times can be substantially reduced. Many file systems have two special requests, OPEN, to copy a specific file directory entry into main memory, and CLOSE, to indicate the entry is no longer needed in main memory. Likewise, we talk about a file being open or closed depending upon the current location of its file directory entry.

5.3.2 Symbolic File System

The first processing step is called the Symbolic file System (SFS). A typical call would be :

CALL SFS (function, file name, record number, location), such as:

CALL SFS (READ, "ETHEL", 4, 12000)

The Symbolic File System uses the file name to locate that file's unique file direc-tory entry. This function corresponds to step 1 in Figure 5.4.

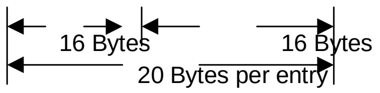
In our simple system we had assumed a one-to-one correspondence between me names and files. In more advanced file systems, the facility must exist to have several files with the same name and be able to call the same me by different names. To implement such facilities, we divide the file directory of Figure 5-3 into two separate directories, a Symbolic File Directory (SFD) and a Basic File Directory (BFD) as shown in Figure 5-5. The symbolic file system must search the symbolic file direc-tory to determine a unique identifier (10) for the file requested (see Figure 5-5). This ID is used instead of the file's name to locate entries in the Basic File Directory. The Basic File System allows operations on files using the 10 such as :

CALL BFS (READ, 5, 4, 12000)

Where 5 is ETHEL's 10.

Since the SFDs are usually kept on the storage device, entries for files that are currently in use (called active or opened files) are copied into main memory. These copies can be used to eliminate repeated 1/0 access to the same entry on the storage device. We will call this table of active me entries the Active Name Table (ANT).

	Name	Unique Identifier (ID)
1	MARYLIN	3
2	ETHEL	5
3	JOHN	6



(a) Sample Symbolic file directory

ID	LOGICAL RECORD SIZE	NUMBER OF LOGICAL RECORDS	ADDRESS OF FIRST PHYSICAL BLOCK	PROTECTION AND ACCESS CONTROL
1	-----------	------------	0	(Basic file directory)
2	20	3	1	(Symbolic file
3	80	10	2	directory)
4	1000	3	3	Access by everyone
5	500	7	6	(Free)
6	100	30	12	Read only
				Read for MANDIK
7	1000	2	10	Read/ Write for
8	1000	1	15	DONOVAN
				(Free)
				(Free)

(b) Sample Basic file directory

Fig.5.5 Sample Symbolic and Basic File Directory

In summary the Symbolic File System is responsible for :

1. The Symbolic File Directory
2. The Active Name Table
3. Communication with the Basic File System.

The second processing step is called the Basic File System (BFS). A typical call would be :

CALL BFS (function, file 10, record number, location), such as :

CALL BFS (READ, 5, 4, 12000)

The Basic File System uses the file ID to locate that file's entry in the Basic File Directory and copies the entry into main memory. This function corresponds to step 2 in Figure 5.4.

```
DECLARE     1    SFD_ENTRY
                      2 NAME CHARACTER (16)
                      2 ID        FIXED BINARY (31)
DO    I = 1 TO 3
       CALL BFS (READ, 2,1,SFD_ENTRY)
       IF SFD_ENTRY.NAME = DESIRED_NAME
       THEN GOTO FOUND
        END;
FOUND: DESIRED_ID = SFD_ENTRY.ID;
```

By permanently assigning the Symbolic File Directory a specific 10, the SFS can use the facilities of the BFS to search the directory for the desired name and 10. For example, the PL/I program segment Verification (ACV), which processes calls such as

CALL ACV (READ, 2,4, 1200)

Where AFT entry 2 contains the information for me 10 5 (me ETHEL).

In summary, the Basic File System is responsible for:

1. The Basic File Directory

2. The Active File Table
3. Communication with the Access Control Verification module

For the simple file system of Section 5-2, it is quite easy to combine the SFS and BFS functions. In later sections we will introduce additional functionality where this logical separation is much more significant.

5.3.3 Access Control Verification

The third processing step is called Access Control Verification (ACV). A typical call would be

CALL ACV (function, AFT entry, record number, location) such as

CALL ACV (READ, 2,4, 12000)

The Access Control Verification acts as check-point between the Basic File System and the Logical File System. It compares the desired function (e.g., READ, WRITE) with the allowed accesses indicated in the AFT entry. If the access is not allowed, an error condition exists and the file sys-tem request is aborted. If the access is allowed, control passes directly to the Logical File System. This function corresponds to step 3 in Figure 5.4.

5.3.4 Logical File System

The fourth processing step is called the Logical File System (LFS). A typical call would be

CALL LFS (function. AFT entry, record number, location) such as

CALL LFS (READ, 2, 4, 12000)

The Logical File System converts the request for a logical record into a request for a logical byte string. That is, a file is treated as a sequential byte string without any explicit record format by the Physical File System. In this simple case of fixed length records, the necessary conversion can be accomplished by using the record length information from the AFT entry. That is, the

Logical Byte Address = (Record Number - I) X Record Length

and

Logical Byte Length = Record Length

This function corresponds to step 4 in Figure 5.4.

Note that by permanently assigning the BFD file a specific ID (such as ID I in Figure 5-5b) and a specific AFT entry number (such as AFT entry I), the Basic File System can call upon the Logical File System to read and write entries in the Basic File Directory. (It is necessary to have a special procedure for fetching BFD entry 1 into AFT entry 1 when the system is initially started or restarted.)

In summary, the Logical File System is responsible for:

CALL PFS (READ, 2, 1500, 500, 12000)

Note that by permanently assigning the BFD file a specific ID (such as ID 1 in Figure 5-5b) and a specific AFT entry number (such as AFT entry 1), the Basic File System can call upon the Logical File System to read and write entries in the Basic File Directory. (It is necessary to have a special procedure for fetching BFD entry 1 into AFT entry I when the system is initially started or restarted.)

In summary, the Logical File System is responsible for :

1. Conversion of record request to byte string request
2. Communication with the Physical File System

Examples of more complex logical record organizations, such as variable-length records, are presented in later sections.

5.3.5 Physical File System

The fifth processing step is called the Physical File System (PFS). A typical call would be

CALL PFS (function, AFT entry, byte address, byte length, location)

such as

CALL PFS (READ, 2, 1500, 500, 12000)

The Physical File System uses the byte address plus the AFT entry to determine the physical block that contains the desired byte string. This block is read into an assigned buffer in main memory, using the facilities of the Device Strategy Module, and the byte string is extracted and moved to the user's buffer area. This function corresponds to steps 5 and 7 in Figure 5.4.

The mapping from Logical Byte Address to Physical Block Number, for the simple contiguous organization of Figure 5-2, can be accomplished by:

Physical Block Number = Logical Byte Address / Physical Block Size + address of first physical block

For the request for the byte string starting at Logical Byte Address 1500, the Physical Block Number is :

Physical Block Number = 1500 / 1000 + 6 = I + 6 = 7

which is the block containing record 4 of me ETHEL (see Figure 5-2).

The location of the byte string within the physical block can be determined by:

Block Offset = remainder [Logical Block Address / Physical Block Size]

such as :

Physical Block Offset = remainder [1500 / 1000] = 500.

Thus, the byte string starts at the offset 500 within the physical block, as expected from Figure 5.2.

In order to perform these calculations, the Physical File System must know the mapping functions and physical block size used for each storage device. If these were the same for all devices, the information could be built into the PFS routines. Usually there are variations depending upon the device type (e.g., a large disk may be handled differently from a small drum). Thus, this information is usually kept on the storage volume itself and read into the Physical Organization Table (POT) when the system is first started.

If a WRITE request were being handled and the corresponding physical block had not been assigned, the Physical File System would call upon the Allocation Strategy Module (ASM) for the address of a free block of secondary storage.

In summary, the Physical File System is responsible for:

1. Conversion of logical byte string request into Physical Block Number and Offset
2. The Physical Organization Table
3. Communication with the Allocation Strategy Module and the Device Strategy Module

Later sections will introduce many more physical system organizations and performance considerations that have an impact on the physical file system.

5.3.6 Allocation Strategy Module

The Allocation Strategy Module (ASM) is responsible for keeping track of unused blocks on each storage device. A typical call would be:

CALL ASM (POT entry, number of blocks, first block)

such as:

CALL ASM (6, 4, FIRSTBLOCK)

where POT entry 6 corresponds to the storage device on which file ETHEL is to reside and FIRSTBLOCK is a variable in which the address of the first of the four blocks requested is returned.

Figure 5-5 indicates how the location of groups of available blocks can be recorded in the Basic File Directory by treating them as special files. Other techniques for maintaining this information are presented in subsequent sections.

The Device Strategy Module (DSM) converts the Physical Block Number to the address format needed by the device (e.g., physical block 7 = track I, record 3, as shown in Figure 5-2). It sets up appropriate I/O commands for that device type. This function corresponds to step 6 in Figure 5.4. All previous modules have been device-independent. This one is device-dependent. Control then passes to the I/O scheduler.

5.3.7 I/O Scheduler and Device Handler

These modules correspond to the device management routines that were discussed in Chapter 3. They schedule and perform the reading of the physical block con-taining the requested information.

5.3.8 Calls and Returns between File System Modules

After the physical I/O is completed, control is returned to the Device Strategy Module. The DSM checks for correct completion and returns control to the PFS, which extracts the desired information from the buffer and places it into the desired location. A "success" code is then returned back through all other modules of the file system.

Why isn't an immediate return made to the uppermost level? There are two reasons: (I) An error may be detected at any level, and that level must return the appropriate error code (for example, the Logical File System may detect the attempted access of an address beyond the file); and (2) any level may generate several calls to lower levels (for example, the Symbolic File System may call the Basic File System several times to read entries from the Symbolic File Directory).

The Symbolic File System checks to see whether or not a file exists. The Access Control Verification checks' to see whether or not access is permitted. The Logical File System checks to see whether the requested logical address is within the file. The Device Strategy Module checks to see whether or not such a device exists.

5.3.9 Shortcomings of Simple File System Design

Let us summarize some of the assumptions, inefficiencies, and inadequacies of our simple file-system system :

1. **Symbolic File System**

   We assumed that each file had a single unique name.

2. **Access Control Verification**

   How are files shared? How can the same file have different protection rights for different uses?

3. **Logical File System**

   We assumed that each file consisted of fixed-length records, and that these records were viewed as sequential. How are accesses to variable-length records handled? Are there other file structuring methods? Are there more flexible methods for the user?

4. **Physical File System**

   Are there physical structuring other than sequential? Are these more efficient in space? In accesses? We assumed that for each request this module would activate an access to the disk. What is the request following READ ETHEL RECORD (4) were READ ETHEL RECORD (3)? Isn't record 3 in the system buffer area already?

5. **Allocation Strategy**

   How are files stored on secondary storage? Are there fragmentation problems when files are deleted? In other words, is secondary storage being used efficiently?

6. Device Strategy Module

Can this module restructure or reorder requests to match more efficiently the characteristics of the device on which the information is stored?

In the sections that follow we will explore more general techniques for each of these levels, techniques that give the user more flexibility and may thus increase the system's efficiency.

5.3 Check Your Progress.

Fill in the Blanks.

1. Components in General Model of file system areorganized as a ___________ .
2. If the file directory a stored on the volume, then it may take a substantial amount of time to____________ it for each acesses to file.

5.4 SYMBOLIC FILE SYSTEM

The SFS's function is to map the user's symbolic reference to an ID.

1. Multiple Names Given to the Same File

This feature, sometimes called aliases, could be useful if several different programs assumed different names for the same file. The user may also find certain names to be more meaningful under different circumstances.

2. Sharing Files

This feature, sometimes called linked files, gives multiple users access to the same physical file, offering them more flexibility and, frequently, greater system efficiency. This is because buffers in main memory as well as secondary storage space can be shared. If, for example, several users are accessing the same file, only one master copy of that file need exist. An early version of M.I.T.'s CTSS timesharing system provides an example of the inefficiency that can result from a system that does not allow the sharing of files. There was a very popular program called DOCTOR that simulated a psychiatrist. Over half the users on the system had a separate copy of this program. Due to the large size of this pro-gram, there was a time when over half the secondary storage consisted of copies of DOCTOR

3. The Same Name Given to Different Files

Since many users create and use files, it is possible that two different programmers may accidentally choose the same name for a files (e.g., TEST). Thus when one of them uses the control cards as follows:

```
DELETE TEST /* delete my old copy*/
CREATE TEST /* create a new copy*/
```

He may be deleting the other programmer's file TEST. Likewise, the RE

4. Subdividing Files into Groups

A given user (or group of users) may be responsible for a large number of files. These files may logically belong to separate projects or tasks. To help keep track of these files in an organized way, it is desirable to group related files together. Such a group is often called a library (or partitioned data set in IBM terminology).

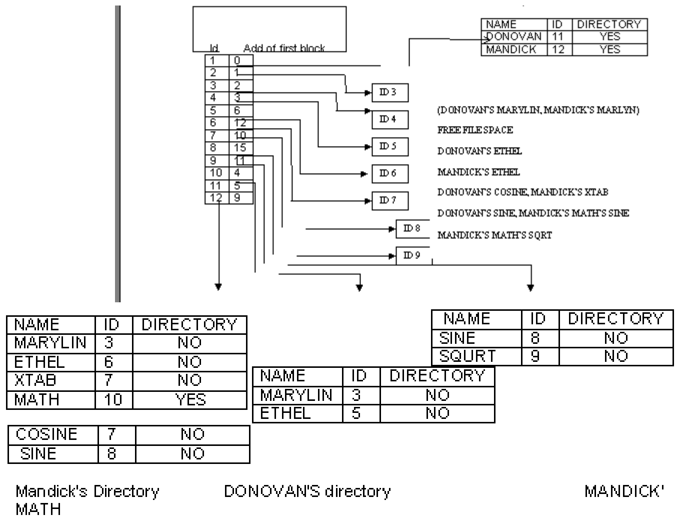
The symbolic file name is a very convenient way for a programmer to remember and refer to this file, but we have noted several shortcomings to the simple me directory scheme presented. We therefore suggest a more flexible me-naming scheme that incorporates all the additional features we have listed above.

5.4.1 Directory Files

Notice that in Figure 5-6 we separated the file directory into two separate com-ponents : the Symbolic File Directory and the Basic File Directory The SFD is concerned with file naming whereas the BFD is concerned with the physical file system.

5.4.2 Sample Hierarchical File Structure

Figure 5-6 illustrates the details of a simple hierarchical file structure consisting of seven data files and four SFD files in addition to the Basic File Directory.



Id	Add of first block
1	0
2	1
3	2
4	3
5	6
6	12
7	10
8	15
9	11
10	4
11	5
12	9

NAME	ID	DIRECTORY
DONOVAN	11	YES
MANDICK	12	YES

NAME	ID	DIRECTORY
MARYLIN	3	NO
ETHEL	6	NO
XTAB	7	NO
MATH	10	YES

NAME	ID	DIRECTORY
SINE	8	NO
SQURT	9	NO

NAME	ID	DIRECTORY
MARYLIN	3	NO
ETHEL	5	NO

COSINE	7	NO
SINE	8	NO

Fig 5.6 Sample hierarchical file structure

The first BFD entry is for the BFD itself; the second entry is always assigned to the Master Directory me. Remember that for all practical purposes it is just like every other file a file directory entry, containing record length, etc. -

The Master Directory file has an entry for each programmer's Symbolic File Direc-tory. In this case, there are two programmers, DONOVAN and MADNICK. Each programmer's directory has entries for all files accessible to him. DONOVAN has four files, MARILYN, ETHEL, COSINE, and SINE, whereas MADNICK has four files; MARILYN, ETHEL, XTAB, and MATH where MATH itself is a Symbolic File Directory with entries for two files, SINE and SQRT.

Notice that the file MARILYN is really the same file (i.e. same ID) for both programmers; this is often called a linked file. On the other hand, DONOVAN's file ETHEL is not the same as MADNICK's file ETHEL (i.e., different IDs). There are several other flexible features illustrated in this example. (Note that DONOVAN's me COSINE and MADNICK's file XTAB are really the same me.) The reader should confirm these observations by studying Figure 5-6.

A hierarchical file organization such as the one described above is often dia-grammed as a tree structure as shown in Figure 5-7. Several file systems (e.g., MULTlCS) allow the programmer to create multilevel subsidiary directories, some-times called private libraries, so that the structure may become several levels deep, as illustrated by the MATH directory in Figures 5.6 and 5.7.

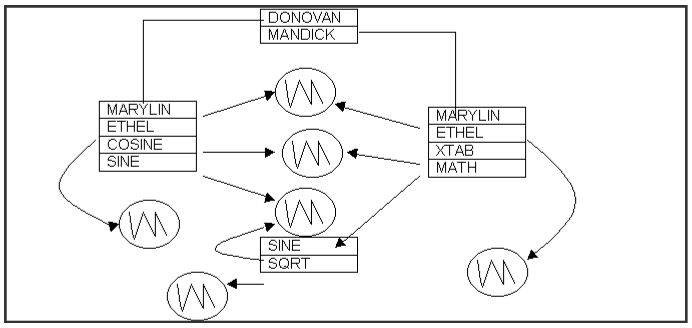


Figure 5.7 A hierarchical file organization

5.4	**Check Your Progress.**

Fill in the Blanks.

1. The Master Directory file has an entry for each programmer's ____________.
2. When the ____________ is called, it is given the unique ID of the file requested.
3. When the file itself is deleted, the ____________ must find and delete all corresponding Symbolic File Directory entries.

5.5 BASIC FILE SYSTEM

When the Basic File System is called, it is given the unique ID of the file requested. The B~S first searches the AFT to determine whether the file is already "opened." If not, the Basic File System searches the Basic File Directory for the entry of the requested file. It then places the information from the entry into the AFT.

If the unique ID assigned to a file is the same as the file's entry number in the Basic File Directory, there is no "searching" required to find the entry in the BFD.

Under certain circumstances, such as when a user wishes to delete a file (e.g., DELETE DONOVAN'S ETHEL), it is desirable to know if this file appears in more than one Symbolic File Directory entry. Frequently, this is accomplished by keep-ing a count in the BFD entry of the number of SFD entries that point to it.

When-ever a link or alias is made, this count is increased. Whenever a link or alias is removed this count is decreased. (When the file itself is deleted, the Symbolic File System must find and delete all corresponding Symbolic File Directory entries.)

5.6 ACCESS CONTROL VERIFICATION

Access control and sharing must be considered in conjunction with each other. For example, a user could protect his information by placing the entire computer system in a locked vault accessible only to him. In some of its installations the government takes an approach somewhat similar to this one by having three separate computer systems-one for top secret information, another for secret information, and yet another for confidential material. Usually, however, this extreme form of protection is not desirable.

In addition to user flexibility, there are several system advantages in allowing a con trolled sharing facility:

1. It saves duplication of files. On one system, for example, there was a single shared copy of a certain 128K program that resided on disk. When the shared copy was removed, about 50 private copies appeared equal to about 64M bytes of disk space.
2. It allows synchronization of data operations. If two programs are updating bank balances, it is desirable that they both refer to the exact same file at all times.
3. It improves the efficiency of system performance, e.g., if information is shared in memory, a possible reduction in I/O operations may occur.

In this section we will suggest three techniques for keeping track of and enforcing access control rights-access control matrix, passwords, and cryptography.

5.6.1 Access Control Matrix and Access Control Lists

Most systems have mechanisms for identifying the user that originates a job. This mechanism is needed to insure correct billing for computer usage. The identifica-tion may be by visual identification by the computer operator or special identifica-tion codes (similar to telephone credit card numbers) submitted as part of the job. We will not pursue this mechanism any further but we will assume that each user can be identified.

An access control matrix is a two-dimensional matrix. One dimension lists all users of the computer, the other dimension lists all files in the system as shown in Figure 5-8. Each entry in the matrix indicates that user's access to that file (e.g., DONOVAN can read file PAYROLL, DONOVAN can both read and write file STOCKPRICE, but MADNICK can read only me STOCKPRICE). The Access Control Verification module compares the user's access request with his allowed access to the file-if they do not match, the access is not permitted.

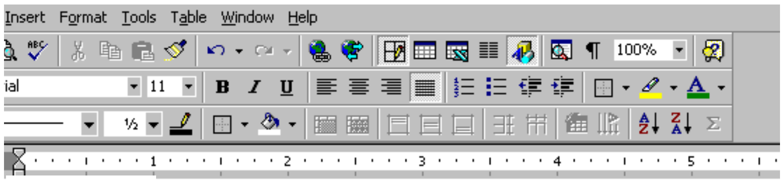


Figure 5.8 Access control matrix

The access control matrix of Figure 5-8 is conceptually simple but poses certain implementation problems. For example, at M.I.T. there are about 5,000 authorized users and about 30,000 online files. A two-dimensional matrix would require 5,000 X 30,000 = 150,000,000 entries. There are various techniques that are logically identical to Figure 5-8 but much more economical to implement. One approach, used in several systems, such as MULTICS, associates an Access Control List (ACL) with each file in the system. The ACL lists the authorized users of this file and their allowed accesses. Logical grouping of users, usually by project, and the special grouping ALL OTHERS greatly reduce the length of the ACL.

For example, if DONOVAN had a file, DATA, which he wished to allow only MADNICK and CARPENTER to read, the ACL would be:

FILE DATA ACL

DONOVAN (ALL ACCESS), MADNICK (READ),
CARPENTER (READ), ALL OTHERS (NO ACCESS)

Note that this required only 4 ACL entries, not 30,000 as would be required for a complete access control matrix. Since most users allow sharing very selectively (public files use "ALL OTHERS" to allow unrestricted access to a large group of users), the Access Control Lists are usually quite short. The ACL information can be efficiently stored as part of the Basic File Directory entry and copied into the Active File Table entry when the file is in use. This makes examination of access control efficient.

5.6.2 Passwords

Access Control Lists, and similar such mechanisms, may take up large amounts of space. Moreover, their maintenance may be complicated, since they may be of varying lengths. Maintenance mechanisms must be implemented either for chaining together all entries of an ACL or reserving blocks large enough to contain the largest ACL.

An alternative method for enforcing access control is through passwords. Associated with each file in the file directory is a password. The user requesting access to that file must provide the correct password. If you wish to allow another user to access one of your files, you merely tell him the password for that file. This method has an advantage in that only a small, fixed amount of space is needed for protection information for each file. Its disadvantage is that a dishonest systems programmer may be able to get all the passwords, since the protection information is stored in the system. This disadvantage also applies to the ACL method.

Another disadvantage of passwords is that access control cannot easily be changed. How does a user selectively retrieve the passwords? If he wishes to deny access of a file to a user who has the password, how does he do it? He can change the pass-word, but now he has to inform all the users who are to be allowed access.

5.6.3 Cryptography

Both the password and the Access Control List methods are disadvantageous in that the "access keys" are permanently stored in the system. Another method is to cryptographically encode all files. All users may access the encoded files but only an authorized user knows the way to decode the contents.

The decoding (for reading) and encoding (for writing) of the file may be done by the Access Control Verification module. The requesting user provides an argument- the code key. The user may change that argument whenever he wishes. There are many techniques for encoding information .One is simply to use the key to start a random number generation. The encoder adds successive random numbers to the bytes of the file to be encoded. Decoding subtracts these numbers to obtain the original data back. An authorized user knows how to start the random number generator with the same code key for decoding as was used for encoding.

In this scheme the code key does not have to be stored in the system. The user needs to enter it only while he is encoding or decoding the file. Thus, there is no table that a dishonest systems programmer can read to find all code keys or allow himself access. Cryptography has the advantage of not having the key stored in the system, but it does incur the cost of encoding and decoding of files.

5.7 LOGICAL FILE SYSTEM

The Logical File System is concerned with mapping the structure of the logical records onto the linear byte-string view of a file provided by the Physical File System. In particular, it must convert a request for a record into a request for a byte string.

In conventional file systems and data management systems many additional record structures are supported. These are often called access methods, such as:

1. Sequentially structured fixed-length records
   (sequential and direct access)
2. Sequentially structured variable-length records
   (sequential and direct access)
3. Sequentially structured keyed records
4. Multiple-keyed records
5. Chained structured records
6. Relational or triple-structured records

5.6 & **5.7** **Check Your Progress.**

Fill in the Blanks.

1. The user requesting access to that file must provide the correct ____________.
2. Another disadvantage of passwords is that ________ cannot easily be changed.
3. The decoding (for reading) and ____________ (for writing) of the file may be done by the Access Control Verification module
4. Physical Block Address = ____________ + logical byte address / block size.

5.8 PHYSICAL FILE SYSTEM

The primary function of the PFS is to perform the mapping of Logical Byte Addresses into Physical Block Addresses. In the organization of Figure 5-5, the Logical File System calls the Physical File System, passing to it the logical byte address and length of the information requested. The PFS may first call the Allocation Strategy Module (if a write request) and then the Device Strategy Module, or it may call the DSM directly.

In our division of responsibility, the PFS keeps track of the mapping from Logical Byte Address to the blocks of physical storage. In previous sections the simple mapping function (Physical Block Address = first physical block of file + logical byte address / block size). was a consequence of the simple physical storage structure of the file. All blocks were stored as in Figure 5-2, i.e. contiguous on secondary storage.

5.9 SUMMARY

Information management is concerned with the storage and retrieval of informa-tion entrusted to the system in much the same manner as a library.

5.10 CHECK YOUR PROGRESS - ANSWERS

5.2

1. Container of information
2. Block number
3. File directory

5.3

1. Structured hierarchy
2. Search

5.4

1. Symbolic File Direc-tory.

2. Basic File System
3. Symbolic File System

5.6 - 5.7

1. Password
2. Access control
3. Encoding
4. First physical block of file

5.11 QUESTIONS FOR SELF - STUDY

1. Discuss the simple file system.
2. What you mean by file directory database?
3. Explain with diagram basic file system.
4. Explain the advantage and disadvantages of the password, cryptography and access control.

5.12 SUGGESTED READINGS

1. **Operating System Concepts By** Abraham Silberschatz, Peter B. Galvin & Greg Gagne.
2. **Operating systems By** Stuart E. Madnick, John J. Donovan

❑ ❑ ❑

Notes

Notes

Chapter : 6

Interdependencies : Performance Evaluation

6.0 OBJECTIVES

After Studying this chapter you will be able to-

- Explain performance Evaluation techniques.
- Analyze Tradeoffs between resources.
- State Modeling and analysis of interdependencies.

6.1 INTRODUCTION

Since the scope and number of interdependencies are endless, we cannot -discuss them all. However, we can suggest to the reader an approach, which may help him answer some of the basic questions that may confront him. For example, if you the reader had a computer system and your boss gave you a fixed amount of money to improve system performance, would you buy more core? A faster CPU? More disks? More channels? After reading this chapter, we hope you will have some general ideas, or know how to utilize the literature for help, or be able to perform a mathematical analysis in order to answer your questions.

Let us illustrate the influence of one resource manager on another. If one were to use a memory "management scheme that offers high utilization of memory, such as paging, and the best multiprogramming processor management schemes offering high utilization of processors and low wait time, one would expect to have the best possible system. Not necessarily. The two schemes might interact in such a way as to produce thrashing, a phenomenon in which the processor is spending most of its time in overhead functions instead of executing user process.

We will discuss for each resource manager in turn what is involved in a tradeoff, and then deal with some of the tradeoffs over which the user has control. We will then present an analysis of two aspects' of the independencies of processor and memory management: a swapping versus a paging system. We will also give an analysis of thrashing.

6.2 MEMORY MANAGEMENT

In Chapter 4 we discussed a series of memory management techniques, each of which progressively provided better utilization of memory (e.g., contiguous allocation, partitioned, relocatable partitioned). However, each improvement in memory utilization required more CPU time.

Before choosing a memory management technique, one should analyze the tradeoff of memory versus CPU time. For example, let us assume that each partition in the relocatable partition scheme depicted in Figure 6.1 is 500,000 bytes. Therefore, at a "move" speed of 4,000,000 bytes per second, 125 milliseconds would be required to move one partition. If the majority of jobs on the system were student jobs, it might take longer to move a job than to run it. A typical student job might run for only 100 milliseconds (execution of 100,000 instructions).

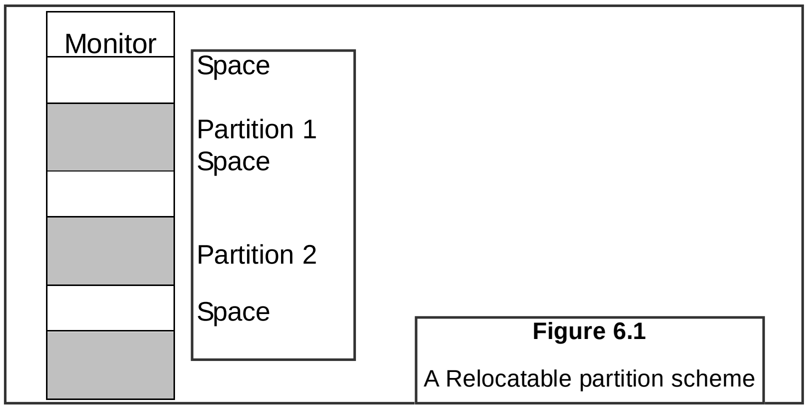


Figure 6.1
A Relocatable partition scheme

6.2 **Check Your Progress.**

Fill in the Blanks.

1. Before choosing a memory management technique, one should analyze the tradeoff of memory versus ___________ .
2. A typical job might run for only ____________.

6.3 PROCESSOR MANAGEMENT

In Chapter 2 we discussed processor management and demonstrated that multi-programming increased processor utilization, but at a cost of memory and I/O device loading.

Figure 6-2 gives a typical curve for the count of CPU time (I/O) waiting for I/O as a function of the number of jobs multiprogrammed in the system. We have assumed that all jobs are equal, and, when run alone, that each job results in the same fixed amount of I/O wait time. Data for a particular group of jobs showing percent of I/O waits can be found in table.

This overhead results from two factors :

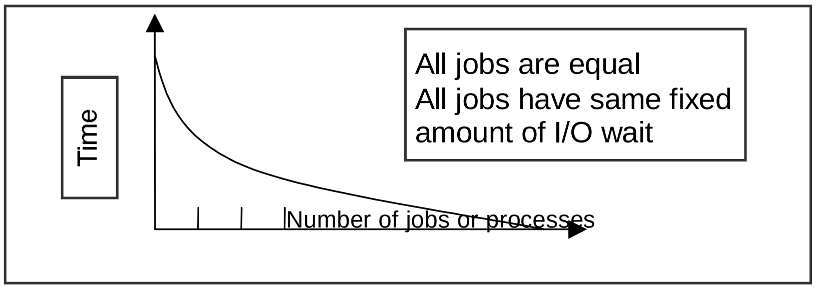


Figure 6.2

1. The I/O queuing requires CPU time. This is usually a constant amount, e.g., 1 ms, but for large numbers of jobs the overhead time grows.
2. There are only a finite number of channels, control units, and I/O devices. Eventually contention develops. Thus for large numbers of processes the CPU may become increasingly more idle as processes wait for busy devices.

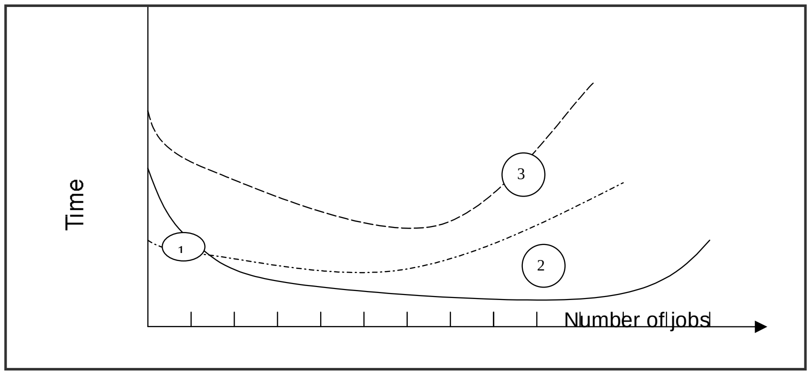


Figure 6.3 Unusable CPU time as a function of increased multiprogramming where
1. I/O saturation
2. T CPU spent in queuing
3. Tunable = T I/O + T cpu queuing

Figure 6.3 shows the total CPU time that is not usable (time spent waiting for I/0 or in the overhead of maintaining queues). Note that after a certain degree of multiprogramming system performance may decrease. (This example is not taken from a paging system; paging may further degrade performance when too much multiprogramming is allowed.) For any given system, there is it point at which increased multiprogramming has a negative effect both on memory and I/O device loadings.

6.3 Check Your Progress.

Fill in the Blanks.

1. The ___________ queuing requires CPU time. This is usually a constant amount.
2. For any system, there is a point at which increased multiprogramming has a ____________ effect both on memory and I/O device loadings.

6.4 DEVICE MANAGEMENT

In Chapter 3 we discussed several techniques for improving the management of devices (e.g., queue reordering, buffering, blocking, SPOOLING). How effective are these techniques in improving system performance? For many systems, such elaborate techniques may introduce more overhead than they can possibly save in efficiency .An example would be a system in which two jobs were multiprogrammed, each with a 25 percent I/0 wait. How much time is the CPU idle? Table 6.4 indicates that the CPU is idle 4 percent of the time. In such a system, how much savings can the cleverest device management techniques provide us? Since multiprogramming reduces the I/O wait time to 4 percent, we can only hope to further reduce this wait to zero! Thus any device management technique that consumes over 4 percent of the CPU time in overhead is costing more in overhead than it can possible save in efficiency.

The reader should not infer, however, that this argument applies to all systems inasmuch as many jobs have much more than 25 percent I/0 wait time.

6.4.1 Queue Reordering

In this section let us examine the tradeoffs involved in the use of the following techniques :

Queue reordering
Blocking/buffering
Data packing techniques

6.4.2 Queue Reordering

Let us take some numbers to illustrate when queuing may not be appropriate. Consider the range of CPUs and their typical instruction speeds. Now consider several disks and their corres-ponding access speeds (Fig. 6-5)- Note that the access speed is the time between a START I/0 to a device and the I/O complete from that device. During this time the CPU may have to wait. Take, for example, the access speed of the 2314, which is 75 ms; this is the maximum time (worst case) that the CPU may be idle. In practice, in a multiprogrammed environ-ment, the CPU would not wait that long because it would most probably execute another process.

COMPUTER	INSTRUCTION (MICROSECONDS}	INSTRUCTIONS (MILLISECONDS}
360/30	30	30
360/40	10	10
360/50	3	3
360/65	1	1
370/15	8	0.4
370/16	8	0.2

Figure 6.4 Instruction speed for computers

Disk Access	Time
IBM 2311	125
IBM 2314	75
IBM 3330	35
IBM 2301	10
IBM 1305	5

6.5 Disk Speed Where first three are moving arm and last two are drumlike

A typical queuing routine may execute 1,000 instructions. Thus on the 360{30, it would take longer to queue than to randomly access the 2301 or 2305, and it would be close on the 2314. On the 360{65, even for the 2305 it is worthwhile to queue if there is a substantial I/O load. Note: if the I/O load is moderate and we are multiprogramming, the I/O might be all overlapped-thus the queuing time would be all CPU overhead.

6.4.3 Blocking/Buffering

Blocking and buffering are device management techniques for reducing the number of I/Os and making better use of I/O device capacity.

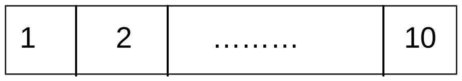
However, the costs incurred are more memory for the buffers, and more CPU time to deblock. On many systems, the memory could be better used for additional multiprogramming than for extra buffers, and then perhaps I/O device capacity would not be a concern. For example, at 4K per buffer, two double-buffered files (input and output) equal 16K. This might equal or exceed the size of a typical "small" job (360/308 go up to only 64K).

6.4.4 Data Packing Techniques

In the following example, each of the three record-packing techniques offers a tradeoff in CPU time, I/O accesses, or memory. Our example has the typical charac-teristics of a card SPOOLING situation where: (1) output spool lines are only ~ 80 to 132 bytes; and (2) of the 80 bytes average, about 50 percent are blanks, often in sequences.

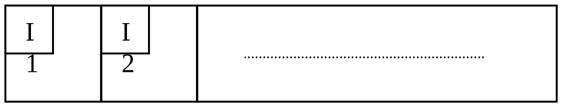
Technique 1

Always write 132-byte records (assume disk < block = 1320 bytes) = 10 records per block.



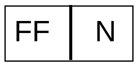
Technique 2

Truncate trailing blanks and write variable length records (average 16 records per block), thus reducing I/O time.



Technique 3

Encode all sequences of multiple
belong here for n > 2),



blanks (e.g., 2 bytes means n blanks

Possibly up to 20 records per block, thus reducing I/O time but increasing CPU time.

The reader should attempt to answer the following questions:

1. Can technique 2 ever require more I/Os than I?
2. How about 3 versus 2?
3. What about block size and number of buffers?

a. (10 users running) X (2 input + 2 output) = 40 buffers

b. (40 buffers) X (4000 bytes per buffer) = 160,000 bytes for buffers [if only 1 buffer per spool space (80K), are added buffers worth 80K?]

In a system designed by one of the authors, technique 2 was chosen because the system was CPU-limited.

6.5 **Check Your Progress.**

Fill in the Blanks.

1. The four techniques for saving secondary storage space are blocking, multime blocking, encodng and ___________ compacting.
2. ____________ blocking is grouping of files together, which eliminates wasted space due to "file breakage".

6.5 INFORMATION MANAGEMENT

The secondary storage allocation techniques focused mostly on flexibility for the user (e.g., dynamic allocation enabled the user files to" grow). However, the possibility of saving secondary storage space itself should be considered. The following are four techniques for saving secondary storage space, each of which costs CPU time.

1. Blocking-grouping of logical records.
2. Multime blocking-grouping of files together. This eliminates wasted space due to "file breakage." (File lengths are seldom-even multiples of physical block length.)
3. Encoding-for example, replacing strings of zeros by a code and a number indicating the number of zeros.
4. Semantic compacting-for example, the statement

6.6 TRADEOFFS OVER, WHICH THE USER HAS CONTROL

The following three techniques may be used to increase the speed of algorithms at the cost of more memory.

6.6.1 Tradeoffs Over, Which The User Has Control

The following three techniques may be used to increase the speed of algorithms at the cost of more memory.

1. Using hash tables user may use hash-coding techniques to manage, tables that must be searched, thereby assuring fast access. However, to assure this fast access, the hash tables must be sparsely filled (e.g., half-filled). Thus it may take twice as much space to store information using hashed tables as it does to store it in tables suited for binary search techniques.
2. Tabulating functions user may use tables instead of computing functions (e.g., store all needed sine values rather than computing them each time).
3. Reducing or eliminating I/O-a user may keep his information in core. This is a direct tradeoff of core space for I/0s.

6.6 Check Your Progress.

Fill in the Blanks.

1. Every container has a _____________that determineshow different components are positioned within the container.

2. A __________ is fairly generic Container.

3. For Frame default LayoutManager is ________

6.7 INFLUENCES

In order to take advantage of techniques for saving CPU time or space the user must be aware of some of the operating system's algorithms or else he may get "out of synchronization" and actually degrade system performance. Naturally, he must also be aware of the cost of each resource before he makes any tradeoffs.

The user should also take into account his own temperament; if for example, he is impatient, he will demand fast turnaround, no matter what the cost.

6.7.1 Influence of a Multiprogrammed System on the User

It is important for a user to know whether his system is multiprogrammed or not. For example, if the system is not multiprogrammed, it is common to have jobs of 100K and core of 256K; in such systems space is not usually a problem. Therefore, a user may freely use more space to save CPU time. Figure 6.6 depicts the performance curve of a user's algorithm as it uses greater space. (e.g., more buffers). As space becomes more plentiful, the user can turn to hashing or similar table-processing techniques, thus reducing his execution time.

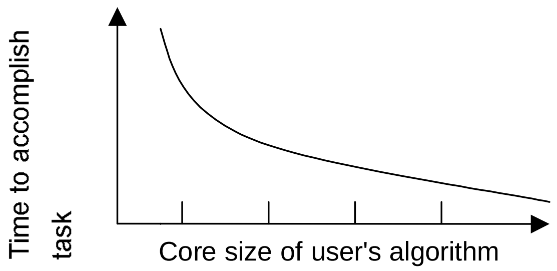


Figure 6.6 Possible user program performance

On a multiprogrammed system, however, if one user increases his core usage, the system cannot multiprogram as many users, which could reduce system perform-ance. Figure 6-7 depicts a typical curve for system performance (throughput) for a fixed core size in a multiprogrammed environment. Point 1 of Figure 6-7 represents a situation in which each job in the system has a small amount of core. If each job were given more core, system performance could be improved, e.g., by reducing I/O (point 2).

However, as users use more and more core, other users are excluded, thus re-ducing multiprogramming (point 3) and degrading system performance. The best size core is at point 2, but this would vary with the types of jobs in the system.

Some measurements of costs have been taken on the user tradeoff of I/O access.

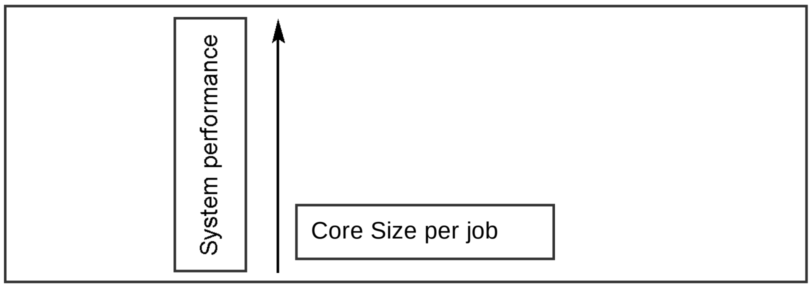
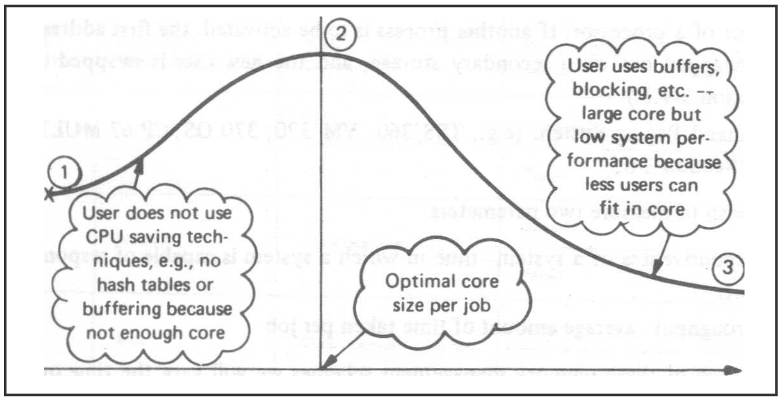


Figure 6.7

System performance in a multiprogramming environment as users use more and more core versus CPU time. On a fully loaded CP-67 it was found that 12 ms of CPU time equalled one I/0 device access That is, if a user could (by buffering, blocking, etc.) spend less than 12 ms of CPU time and save one I/O, he would come out ahead.



One tradeoff over which a user has less control showed that II ms of user CPU time equals the cost involved in 1.4 page I/Os. Thus if a user could perform more CPU calculations and thereby reduce his paging requirements, he would reduce costs. Since paging is (or should be) invisible to the user, this equality is not readily usable.

6.7.2 Psychological Tradeoffs

Recently, the authors were involved in providing a financial reporting package for a small company. Since the reports were to be printed in alphabetical order but not stored alphabetically in the database, there were two possible algorithms:

1. For each item, search the database and print the line.
2. Perform a sort on the database and print the entire sorted database.

The second was the more efficient for the database size, but one of the authors could not stand the suspense of having the computer spend a considerable amount of time in internal sorting before receiving answers. He thus reprogrammed the algorithm in the inefficient fashion of the first algorithm.

6.7 **Check Your Progress.**

Fill in the Blanks.

1. In order to take advantage of techniques for saving ___________ or ___________ the user must be aware of some of the operating system's algorithms.

2. Therefore, a user may freely use more space to save ___________ time

3. On a multiprogrammed system, however, if one user increases his ___________, the system cannot multiprogram as many users.

6.8 THRASHING

In the last section we saw the advantages of paging. We have also discussed the advantages of multiprogramming. If we combine these two techniques, would we have too much of a good thing? Paging is a memory management function, while multiprogramming is a processor management function. Are the two interdepen-dent? Yes, because combining them may lead to a drastic degradation of system performance. This degradation is loosely called thrashing. Among the definitions of thrashing that have been used are the following:

1. Thrashing is the point at which a system drives its disks, at their natural resi-dency frequency; into doing so much I/O that the disk looks like a washing machine in its drying cycle.

2. Thrashing is the region in which the system is doing a lot of work but with no results.

3. Thrashing is the point at which the total CPU time devoted to a job in multi-programming is equal to the total CPU time that would be devoted to that job in a monoprogrammed environment, that is, the point at which multiprogramming loses to monoprogramming.

6.8 **Check Your Progress.**

Fill in the Blanks.

1. Thrashing is the point at which a system drives its disks, at their ___________.

2. Thrashing is the point at which the total CPU time devoted to a job in ___________is equal to the total CPU time that would be devoted to that job in a monoprogrammed environment.

6.9 SUMMARY

Taking the "best" memory processor, device, or information management schemes and combining them all into one system will not necessarily produce the "best" system. We have touched on a few of the tradeoffs that a system designer and/or user may wish to use to make a system better suited to his needs. We have presented analytical techniques for measuring these tradeoffs and providing the information necessary to make system decisions.

6.10 CHECK YOUR PROGRESS - ANSWERS

6.2

1. CPU time 2. 100 milloseconds

6.3

1. I/O queuing 2. hegative

6.4

1. 4 2.buffering

6.5

1. Semantic 2. MULTIME

6.6

1. Hash table 2. I/O 3. Computin

6.7

1. CPU time or space 2. CPU

6.8

1. Natural residential frequency. 2. Multiprogramming

6.11 QUESTIONS FOR SELF - STUDY

1. Describe memory, process and device management.
2. Explain Influence of a multiprogrammed system on the user.
3. What do you mean by Psychological Tradeoffs?
4. Discuss various definitions of thrashing.

6.12 SUGGESTED READINGS

1. **Operating System Concepts By** Abraham Silberschatz, Peter B. Galvin & Greg Gagne.
2. **Operating systems By** Stuart E. Madnick, John J. Donovan

❑ ❑ ❑

Notes

Chapter : 7

File System

7.0 OBJECTIVES

After studying this chapter you will be able to -

- Distinguish between windows file system and Linux file system.
- Discuss the concept of Data Backup.

7.1 INTRODUCTION

File system is a structure of data storage. i.e. how the data is stored in the form of files, how directories are created and managed with certain level of security. Different operating systems have different file systems. Files and folders (directories) are stored on hard disk as a storage device.

The following are common hard disk configurations.

- **Partition** - A partition is a portion of a physical hard disk. A partition can be primary or extended
- **Primary Partition -** This is a bootable partition. One primary partition can be made active.
- **Extended Partition -** An extended partition is made from the free space on a hard disk and can be broken down into smaller logical drives. There can only be one of these per hard disk.
- **Logical Drive -** These are a primary partition or portions of an extended partition that are assigned a drive letter.
- **Volume -** This is a disk or part of a disk that is combined with space from the same or another disk to create one larger volume. This volume can be formatted and assigned a drive letter like a logical drive, but can span more than one hard disk. A volume set can be extended without starting over, however to make it smaller, the set must be deleted and re-created.

7.2 WINDOWS FILE SYSTEMS

There are three Windows file systems:

- CDFS--Compact Disc File System; a read-only file system, not configurable

- FAT--File Allocation Table; primarily for compatibility with other OS (dual booting)
- NTFS--New Technology File System; fast, reliable, secure, and robust
- (Also HPFS - High Performance File System - to convert from OS/2)

File Allocation Table (FAT)

Windows 9x uses FAT32, with FAT16 drivers for backward capabilities. Each partition on a hard drive needs to be formatted to hold files (high level formatting) and folders that the OS can access. The FORMAT command creates and configures File Allocation Tables (FAT) and creates the root directory to build the OS upon.

Hard drives store data in areas called sectors, up to 512 Bytes. If you have a file smaller than 512B, the rest of the sector goes to waste. If you have a file over 512B, the OS fills unused, non-sequential sectors until the file is stored completely (this is why you will eventually need to "defrag" every hard drive, too). Once files are written to the disk, the OS needs to remember which sectors have parts of which file: in other words, a File Allocation Table.

FAT16 means the file system uses 16 bits to write addresses (4 hexadecimal numbers: 0000-FFFF). By doing the math, you will see why we outgrew this years ago. Our hard drives got too big: 16 bits equals 65,536 (64K of) sectors. 65,536 (64K) x 512K = 33,554,432 Bytes (or 32,768 KB or 32 MB) TM

To get partitions larger than 32MB with FAT16, we used clusters or file allocation units: contiguous sectors acting like one sector in the FAT table. With between 4 to 64 sectors/cluster, FAT16 was able to see partitions up 2 GB (32K clusters).

With Windows 95, FAT32 was introduced. FAT32 means the FAT table has 32-bit addresses (8 hexadecimal numbers). This also enabled long file names (beyond the 8.3 dos standard). Depending on the cluster size (4-32 sectors/cluster), FAT32 can read partitions up to 2 Terabytes (4K clusters).

Under NT, FAT is actually VFAT, Virtual FAT. The major difference is that hardware is not accessed directly (A virtual device driver "fools" the system into thinking that it is), which contributes to robustness. The file system will not crash with the hardware. It relies upon a linked-list directory structure, which reduces performance on large partitions (each directory entry is linked to the next--the longer the chain, the slower the access) and limits its practical size. A FAT partition is required on RISC machines (for the boot files) and on dual-boot (NT and DOS/Windows systems) machines.

The VFAT used by NT is more flexible than the old DOS FAT--filenames up to 255 characters including the entire path are allowed, periods and spaces in filenames are fine, and case is preserved (without being case-sensitive--FILE.TXT equals File.TXT equals file.txt) Restricted characters are the same-- []',"/;:\=+^*?<>|

VFAT limitations

- No local security (share permissions only)
- Performance begins dropping on partitions over

200MB, especially with large numbers of files

- Performance degrades quickly by file fragmentation
- Theoretical upper limit of 4GB (improved from DOS FAT's 2GB)
- Low fault-tolerance; subject to physical and logical disk errors
- Supports only low-level file attributes (read-only, archive, system, hidden)
- The root directory is limited to 512 entries. Since long filenames are stored in secondary directory entries (one per 13 extra characters beyond 8), too many of those can actually lead to Windows being unable to create files in root, and quite quickly.

Note : There is a command-line-only utility to change a partition from FAT to NTFS - convert.exe. (To convert back to FAT from NTFS requires third-party software not supported by NT) The syntax is: C:\ convert [drive letter/partition]: /FS:NTFS /v

You cannot convert a drive while the drive is being accessed. It may be scheduled for conversion at next bootup, before shares are accessible.

Note : To convert from FAT16 to FAT32, the conversion utility is CVT1.EXE

NTFS

There are two versions of NTFS, NT File System: 4 (NT) and 5 (2000). The following pertain to both.

- ❐ NTFS uses a MFT: Master File Table. NTFS allows you to adjust the sizes of clusters, and can support 2 Terabytes (default) up to 16 Exa bytes.
- ❐ NTFS, as the name implies, is NT's proprietary file system. It supports far more useful characteristics
  - o Fault tolerance
  - o Extended attributes (such as date of creation),
  - o Object-oriented file and directory security (including auditing file use)
  - o File-level compression
  - o Faster locating method (branching tree), and
  - o A truly awesome maximum capacity.
- ❐ NTFS keeps a log of all read/write transactions. For example, if on a SCSI disk, it will automatically scan for bad clusters, mark them as bad and avoid them, and move data to good clusters. If the error is discovered during a write, the data will be rewritten to a sound sector. If a read operation discovers a bad cluster, however, it cannot be read, and the data is lost.
- ❐ Extended attributes were added for POSIX compliance. This includes time stamping for file creation, accessing, and modification. Filenames are case sensitive (e.g. file.txt and File.txt are treated as different and can be stored in the same folder). "Hard links", where the same file is accessed by two different filenames, is also supported.

- Permissions can be set and auditing enabled for individual folders and files. The permissions differ slightly for files and folders, and can be combined with normal share permissions to allow very fine-tuning.
- NTFS supports compression at the drive/partition-, folder/directory-, or filelevel.
- Search functions look for files alphabetically in a branching search. This is much faster than following entries linked in a consecutive chain. Space for new files is allocated dynamically, which means defragmentation is required much less frequently.
- The theoretical maximum size of an NTFS file or partition is 16 Exa bytes. No existing hardware can take advantage of this. An Exa bytes is 260 bytes...or a gigabyte OF gigabytes, for a total of 16 billion gigabytes in a single partition...not a limitation.

NTFS4 Limitations

- Filenames may not contain the characters ><|*"?/:\
- Does not support file encryption
- Works best on partitions over 50MB
- Cannot format 1.44MB floppies because of high overhead (about 5MB per partition) required
- If dual-booting to an OS other than NT, you must have at least one small FAT partition also.
- Must reboot after reformatting a ZIP drive or other removable media (looked all over, and can't find out why)

NTFS4 vs. NTFS5

Windows 2000 uses an updated version of NTFS (sometimes called NTFS5). The major improvements are:

- File and/or directory encryption
- Per-user, per-volume disk quotas
- Hierarchical Storage Management (reparse points)
- Mount points
- Distributed link tracking support
- Sparse-file support
- Volume change tracking
- Encryption.

Windows 2000 Server will require the use of NTFS5 format on all domain controllers.

Similarly, it is likely that most other servers will be using the NTFS5 format to take advantage of the new Windows 2000 Server features.

File Systems

After you create the installation partition, Setup prompts you to select the file system with which to format the partition. Like Microsoft Windows NT 4 and Microsoft Windows 2000 Professional, Windows XP Professional supports the NT file system (NTFS) and file allocation table (FAT). Both Windows 2000 Professional and Windows XP Professional support FAT32. Figure summarizes some of the features of these file systems.

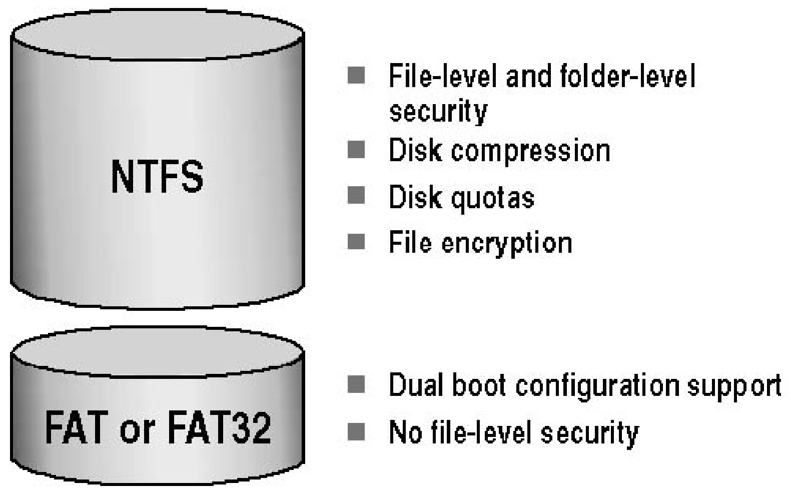


Figure 7.1 NTFS and FAT/FAT32 file system features

Use NTFS when the partition on which Windows XP Professional will reside requires any of the following features:

File- and folder-level security. NTFS allows you to control access to files and folders.

Disk compression. NTFS compresses files to store more data on the partition.

Disk quotas. NTFS allows you to control disk usage on a per-user basis.

Encryption. NTFS allows you to encrypt file data on the physical hard disk, using the Microsoft Encrypting File System (EFS). For additional information.

The version of NTFS in Windows XP Professional supports remote storage, dynamic volumes, and mounting volumes to folders. Windows XP Professional, Windows 2000, and Windows NT are the only operating systems that can access data on a local hard disk formatted with NTFS.

FAT and FAT32

FAT and FAT32 offer compatibility with other operating systems. You must format the system partition with either FAT or FAT32 if you will dual boot Windows XP Professional and another operating system that requires FAT or FAT32.

FAT and FAT32 do not offer many of the features (for example, file-level security) that NTFS supports. Therefore, in most situations, you should format the hard disk with NTFS. The only reason to use FAT or FAT32 is for dual booting with another operating system that does not support NTFS. If you are setting up a computer for dual booting, you need to format only the system partition as FAT or FAT32. For example, if drive C is the system partition, you could format drive C as FAT or FAT32 and format drive D as NTFS.

Converting a FAT or FAT32 Volume to NTFS

Windows XP Professional provides the Convert command for converting a partition to NTFS without reformatting the partition and losing all the information on the partition. To use the Convert command, click Start, click Run, type cmd in the Open text box, and then click OK. This opens a command prompt, which you use to request the Convert command. The following example shows how you might use switches with the Convert command.

```
Convert volume /FS:NTFS [/V] [/CvtArea:filename] [/Nosecurity] [/X]
```

Table 7.1 lists the switches available in the Convert command and describes their functions.

Switch	Function	Required
Volume	Specifies the drive letter (followed by a colon), volume mount point, or volume name that you want to convert	Yes
/FS:NTFS	Specifies converting the volume to NTFS	Yes
/V	Runs the Convert command in verbose mode	No
/CvtArea:filename	Specifies a contiguous file in the root directory to be the placeholder for NTFS system files	No
/NoSecurity	Sets the security settings to make converted files and directories accessible by everyone	No
/X	Forces the volume to dismount first if necessary, and all open handles to the volume are then not valid	No

Table 7.1 Convert Command Switches

For help with any command-line program, at the command prompt type the command followed by /? and then press Enter. For example, to receive help on the Convert command, type Convert /? and then press Enter.

7.1 & **7.2** **Check Your Progress.**

Fill in the blanks

1] File systems is a ___________ of data storage.

2] Different operating systems have ___________ file systems.

3] Theoretical upper limit of 4GB for ___________ partition.

4] NTFS stands for ___________

5] VFAT stands for ___________

State Whether the True or False

1] FAT supports local security.

2] VFAT stands for Very Fast Allocation Table.

3] NTFS stands for New Technology File System.

4] Convert command is used to convert FAT to NTFS.

5] NTFS 5.0 supports disk compression.

6] NTFS 4.0 supports user quota and disk quota.

7.3 LINUX FILE SYSTEM

Linux is a network operating system. It is a plug and play 32 or 64 bit operating system with GUI and Command Line Interface (Console Mode). Unlike Windows operating systems it supports two GUI interfaces GNOME (GNU Network Object Model Environment) and KDE (K Desktop Environment). Linux is based on UNIX operating system. Most of the Linux commands are similar to UNIX commands. It provides various network services like file services (NFS- Network File System), Print Services (CUPS- Common Unix Print Services), Web services (Apache Web services), Mail services (Send Mail Services), Data Base services (MY SQL), Proxy services (Squid Proxy), FTP services (VSFTPD- Very Secure File Transfer Protocol Daemon), DHCP services (Dynamic Host Configuration Protocol) etc.

Linux supports three file systems which are as follows ext2 file system , ext3 file system and swap file system. It also supports RAID (Redundant Array of Independent or Inexpensive Disks) and LVM (Logical Volume Manager) as file systems.

Linux File System Concepts

1] In Linux every thing is a file including hardware.
2] Files and directories are organized into a single rooted inverted tree structure.
3] File system begins at the "root" directory represented by a lone "/" (forward slash) characters.
4] File names are case sensitive. For e.g. file1, File1 and FILE1 are three different files.
5] All characters are valid except / for file names.
6] Maximum file name is up to 255 characters.
7] Ext3 file system support User Quota and Disk Quota.
8] Journaling technique is used to check the integrity of the files.

7.3 Check Your Progress.

Fill in the Blanks.

1] Linux is a __________ operating system.
2] Linux is a __________ operating system.
3] Linux supports __________ as well as __________
4] LVM stands for __________
5] RAID stands for __________
6] GNOME stands for __________
7] KDE stands for __________

State whether the true or false

1] Linux is a desktop operating system.
2] Linux supports two GUI interfaces GNOME and KDE.
3] In Linux every thing is a file including a hardware.
4] In Linux file names are not case sensitive.
5] In Linux passwords are case sensitive.
6] In Linux file name is up to maximum 300 characters.

7.4 DATA BACKUP

In Information Technology , a backup or the process of backing up refers to making copies of data so that these additional copies may be used to restore the original after a data loss event. These additional copies are typically called "backups.

Backups are useful primarily for two purposes. The first is to restore a state following a disaster (called disaster recovery). The second is to restore small numbers of files after they have been accidentally deleted or corrupted. Data loss is also very common. 66% of internet users have suffered from serious data loss.

In the modern era of computing there are many different types of data storage devices that are useful for making backups. There are also many different ways in which these devices can be arranged to provide geographic redundancy, data security and portability. Many different techniques have been developed to optimize the backup procedure. These include optimizations for dealing with open files and live data sources as well as compression, encryption, and de-duplication, among others.

Why do you need backup?

Backups are needed in case a file or a group of files is lost. The reasons for losing files include Hardware failure like disk breaking, accidentally deleting wrong file and computer being stolen. Backups help in all the above situations. In addition, it may be good to have access to older versions of files, for example a configuration file worked a week ago, but since then it has been changed and nobody remembers how, its just not working anymore.

Backup devices and media

You need some media to store the backups. It is preferable to use removable media, to store the backups away from the computer and to get "unlimited" storage for backups.

If the backups are on-line, they can be wiped out by mistake. If the backups are on the same disk as the original data, they do not help at all if the disk fails and is not readable anymore. If the backup media is cheap, it is possible to take a backup every day and store them indefinitely.

Floppy, Hard Disk, Tapes, CD-R,CD-RW, DVD, DVD -RW are the medias available for backup.

DAT (Digital Audio Tape) Drives, DLT(Digital Linear Tape) Drive, and LTO (Linear Tape Organizer) Drive these are tape devices used to take the backups. Most of the tape devices are having SCSI interface but now a days USB 2.0 tape devices are also available.

Types of Backup

There are different kinds of backups, the following lists some of them:

Full Backup

Full backup is the starting point for all other backups, and contains all the data in the folders and files that are selected to be backed up. Because full backup stores all files and folders, frequent full backups result in faster and simpler restore operations. Remember that when you choose other backup types, restore jobs may take longer.

Advantages

Restore is the fastest

Disadvantages

Backing up is the slowest

The storage space requirements are the highest

Incremental Backup

Incremental backup means backing up everything that has changed since last full backup.

Advantages

Backing up is the fastest

The storage space requirements are the lowest

Disadvantages

Restore is the slowest

Differential Backup

Differential seems to be another name for incremental. differential backup offers a middle ground by backing up all the files that have changed since the last full backup

Advantages

Restore is faster than restoring from incremental backup

Backing up is faster than a full backup

The storage space requirements are lower than for full backup

Disadvantages

Restore is slower than restoring from full backup

Backing up is slower than incremental backup

The storage space requirements are higher than for incremental backup

Network Backup

Network backup usually means backing up a client to a backup server, this means the client sends the files to the server and the server writes them to backup medium.

Dump Backup

Dump backups are not ordinary file by file backups. The whole disk partition or file system is "dumped" to the backup medium as is. This means it is also necessary to restore the whole partition or file system at one go. The dump backup may be a disk image, which means it must be restored to a similar disk with same disk geometry and bad blocks in same places.

Choosing a Backup Tool in Linux

Linux has several tools for backing up and restoring files dump / restore : Old tools that work with file systems, rather than files, and can back up un mounted devices. Although you can easily control what is backed up with dump by editing a single column in the /etc/fstab file, for some reason these utilities have fallen into disuse. Today, many distributions of Linux, including Debian, do not even include them by default. If you want to use dump and restore , you must install them yourself.

tar : A standard backup tool, and by far the easiest to use. It is especially useful for backing up over multiple removable devices using the -M option.

cpio : A very flexible command, but one that is hard to use because of the unusual way in which the command must be entered.

dd : The dd command is one of the original Unix utilities and should be in everyone's tool box. It can strip headers, extract parts of binary files and write into the middle of floppy disks; it is used by the Linux kernel Makefiles to make boot images.

Mondo : Mondo is reliable. It backs up your GNU/Linux server or workstation to tape, CD-R, CD-RW, DVD-R[W], DVD+R[W], NFS or hard disk partition. In the event of catastrophic data loss, you will be able to restore all of your data [or as much as you want], from bare metal if necessary. Mondo is in use by Lockheed-Martin, Nortel Networks, Siemens, HP, IBM, NASA's JPL, the US Dept of Agriculture, dozens of smaller companies, and tens of thousands of users.

Dar: dar is a shell command that backs up directory trees and files. It has been tested under Linux, Windows, Solaris, FreeBSD, NetBSD, MacOS X and several other systems.

Many commercial or free software back up tools are also available.

7.4 Check Your Progress.

Fill in the blanks

1] Additional copies of data is called ___________

2] Backups are generally taken on ___________ medias.

3] In full backup the speed of backup is ___________

4] In incremental backup backing up is ___________

5] ___________ backup restore is slowest.

6] Dump utility is used in Linux to take backup of a ___________

7.5 SUMMARY

Partition is portion of physical hard disk. File System is a structure of data storage,

FAT primarily for compatibility with other Operating System Windows gx uses FAT 32.

Windows XP professional provides the convert command for converting a partition to NTFS without reformatting partition and losing all information on partition.

Linux is network Operating System. It is plug and play 32 164 bit operating system with GUI and command line interface.

The process of backing up refers to making copies of data so that these additional copies may be used to restore the original after a data loss event. These copies are called backups.

Source : *www.scribd.com › Research › Internet & Technology(Link)*

7.6 CHECK YOUR PROGRESS - ANSWERS

7.1

1] Structure

2] different

3] VFAT

4] New Technology File System.

5] Virtual File Allocation Table.

7.2

1] False

2] False

3] True

4] True

5] True

6] False

7.3

1] Network

2] Plug and Play

3] Graphical User Interface and Command Line Interface (Console)

4] Logical Volume Manager

5] Redundant Array of Independent Disks.

6] GNU Network Object Model Environment.

7] K Desktop Environment.

State true or false

1] False 2] True 3] True 4] False 5] True 6] False

7.4

1] Backup	2] Different
3] Slowest	4] Fastest
5] Incremental	6] Partition.

7.7 QUESTIONS FOR SELF - STUDY

1. Explain various hard disk configurations.
2. Explain FAT in detail.
3. Explain NTFS and FAT / FAT 32 File system features in details.
4. Explain Linux File System in detail.
5. Explain type of backup in detail.

7.8 SUGGESTED READINGS

1. File systems: design & implementation by Daniel Grosshans
2. Practical File System Design with the Be File System By Dominic Giampaolo Be, Inc.

❑ ❑ ❑

Notes